Theoretical Computer Science 93 (1992) 1–41
Elsevier

Fundamental Study

On the expressive power of finitely typed and universally polymorphic recursive procedures*

A.J. Kfoury

Boston University, Computer Science Department, 111 Cummington Street, Boston, MA 02215, USA

J. Tiuryn**

University of Warsaw, Institute of Mathematics, PKiN, 00-901 Warszawa, Poland

P. Urzyczyn

University of Warsaw, Institute of Mathematics, PKiN, 00-901 Warszawa, Poland

Communicated by M. Nivat
Received December 1989

Abstract

Kfoury, A.J., J. Tiuryn and P. Urzyczyn, On the expressive power of finitely typed and universally polymorphic recursive procedures, Theoretical Computer Science 93 (1992) 1–41.

Finitely typed functional programs are naturally classified by their levels. This syntactic classification of functional programs corresponds to a semantical classification: the higher the level of functional programs, the more functions they can compute. We call *FL* the language of finitely typed functional programs. The halting problem on finite interpretations is elementary recursive for every *FL* program, i.e. for every *FL* program *P* there is an elementary recursive procedure to decide for every finite interpretation $\mathcal{I}$ whether *P* halts on $\mathcal{I}$.

The well-known programming language *ML* is essentially *FL*, augmented with the polymorphic **let-in** constructor. We show that *ML* computes the same class of functions as *FL*. As a consequence,

* This research was partly supported by NSF grant CCR-8901647 and by a grant of the Polish Ministry of National Education, no. R.P.I.09. Some of the results in this paper have been presented at the Symposiums "Logic in Computer Science" in 1987 and 1988 (see [17] and [18]).

** Part of the work was done when the author was visiting Computer Science Department of Washington State University, Pullman, WA.

the presence of the polymorphic **let-in** constructor is not the source of additional computational power, and the halting problem on finite interpretations for every *ML* program is also elementary recursive. On the other hand, we show that if *ML* is augmented with a polymorphic fixpoint constructor, the resulting extension *EML* of *ML* computes more functions. In fact, we show that *EML* has universal computational power, which implies the undecidability of the halting problem on finite interpretations for *EML* programs.

Contents

0. Introduction

Many modern programming languages allow their programs to use higher-order objects in their computations. The role of such objects is purely auxiliary in that, semantically, a program is seen as transforming ground values into ground values, i.e. higher-order values are not mentioned in the input–output relation of the program. Moreover, depending on whether the language is finitely (i.e. rigidly) typed or polymorphically typed, higher-order objects are required to have the same type in every activation or are allowed to have different types from one activation to another.

Our finitely typed programming language *FL* is essentially the finitely typed λ-calculus, with first-order constants and recursion. Recursion is introduced in one of two equivalent ways: either by means of a fixpoint constructor **fix** or by means of mutually recursive function definitions. The main function definition in a functional program is always first-order, i.e. the function is applied to input values of ground type and returns an output value (if any) of ground type, but it can call intermediary defined functions of arbitrary finite orders. This is essentially the language PCF described in [33], except that we do not interpret programs in arithmetic, and leave them as "program schemes" that are compared over all possible interpretations. Relative to a single interpretation we use a call-by-name operational semantics, which we may define by structured rules in the style of [33, 34].

The finitely typed functional programs are naturally classified by their order or "level", so that we have a natural hierarchy of languages:

$$FL_0 \subset FL_1 \subset FL_2 \subset \cdots \subset \bigcup_{n \in \omega} FL_n = FL.$$

The level n of a functional program is the highest order of a function definition used in the program.

What has been often called a "core" fragment of ML (see e.g. [2, 4, 29]) is our language FL, enriched with the polymorphic **let-in** constructor. Enriching this core ML with universally polymorphic recursion, we call EML the resulting extension of ML. Universally polymorphic recursion is introduced by an appropriate typing of the fixpoint constructor **fix** (precisely defined in Section 5). Basically, universally polymorphic recursion allows the actual parameters of recursively called functions to have types that are generic instances of the derived types of the corresponding formal parameters. This feature is not allowed in standard ML, where **fix** is monomorphically typed, forcing actual parameters to have types that are equal to the derived types of the corresponding formal parameters.

There are important pragmatic reasons for the introduction of the polymorphic **let-in** and the polymorphic **fix**. A discussion of these reasons is beyond the scope of this paper. For the polymorphic **let-in**, the reader may wish to consult any of the several well-known references on ML and ML-like languages (see, e.g. [2, 29, 32]). The justification of the polymorphic **fix** from the point of view of programming language design can be found in [13, 19, 20, 28, 30].

The aim of this paper is, first, to study the impact of restricting the maximal level of higher-order objects used by FL programs on the computational power of such a class of programs. Second, we examine the effect on the computational power of adding the polymorphic **let-in** to obtain ML. Third, we examine the effect on the computational power of adding the polymorphic **fix** to obtain EML. The expression *computational power of a class* $\mathscr{P}$ *of programs* should be understood as the collection

$$COMP(\mathscr{P}) = \{COMP(\mathscr{P}, \mathscr{A}) \mid \textit{arbitrary } \mathscr{A}\},$$

where $\mathscr{A}$ ranges over all structures in the same first-order signature (the universe of each being a set of ground values) and $COMP(\mathscr{P}, \mathscr{A})$ is the set of all input–output relations computed in $\mathscr{A}$ by programs of $\mathscr{P}$. The main results of this paper are:

- The family $\{FL_n\}$ forms an infinite hierarchy of more and more powerful programming languages. This result is obtained by applying the method of *spectral complexity measures* introduced in [36] and developed further in [35]. We use the results of [17] together with the recent improvement of [12] to establish the strictness of the hierarchy.
- The language ML is no more powerful than FL, so that, in particular, the halting problem on finite interpretations for every ML program is elementary recursive. This result was previously obtained in [18].

- The language *EML* has universal computational power. In particular, the halting problem on finite interpretations of *EML* programs is undecidable. [18] contains a proof that this problem is at least primitive recursive, leaving as an open problem whether it is decidable.

In summary, increasing the level of finitely typed functional programs results in a gain of computational power, adding the polymorphic **let-in** does not, and adding the polymorphic **fix** results in a maximal gain of the computational power. Hence,

$$FL_0 < FL_1 < \cdots < \bigcup_{n \in \omega} FL_n = FL \equiv ML < EML.$$

To put the results of this paper in a proper perspective, we note that the notion of computational power here is closely related to the natural notion of *semantical equivalence* of programs and, therefore, it captures the "real" computational power of programs in all abstract structures. It should not be confused with other, usually stronger, notions of program equivalence, e.g. *formal equivalence* (cf. [27]). Two programs are said to be *formally equivalent* if their reduced formal computation lattices are the same; program equivalence in this sense leads to the comparison of certain formal languages associated with program schemes. With respect to this latter notion of program equivalence, the study of recursion on higher types as a control structure for programming languages (both imperative and functional) is not new. It can be traced at least as far back as [15, 29, 33]. With an interest more in formal languages than in programming languages, Damm, Engelfriet and several of their colleagues studied various hierarchies which appear related to ours (see [6, 7, 8, 9], among others). An important paper of Damm [5] establishes the strictness of the hierarchy of (finite-mode) ALGOL-68 program schemes with respect to the maximal level of types (i.e. modes) of procedures; however, this strictness result is merely related to the notion of formal equivalence of programs. Thus, the question of whether the hierarchy is strict with respect to the usual notion of program equivalence was left open.

We illustrate the difference between the results of [5] and the questions we ask in this paper: there are two distinct senses in which one can justify the statement "recursive procedures (with ground type parameters) are more powerful than iterative programs". In the first sense, corresponding to formal equivalence, one justifies the statement by observing that the context-free languages form a richer family than the regular languages. In the second sense, related to the usual notion of program equivalence, one justifies the statement by using a "pebble-game" argument ([10, 31]) to show that there exists an interpretation in which recursive procedures compute *more functions* than iterative programs. Since the formal equivalence of programs is a stronger notion than the semantical equivalence, it follows that strictness of the hierarchy with respect to the latter notion implies strictness with respect to the former notion.

The paper is organized as follows. Section 1 states some preliminary definitions and results. In Section 2 we consider an untyped language (essentially the λ-calculus

augmented with additional constructs) which provides base syntax and semantics rules for all our languages, and we prove several lemmas which are necessary later. Section 3 discusses the finitely typed languages FL_n. In Section 4 we prove that the expressive power of ML is equal to that of FL, and in Section 5 we show that EML is of universal power.

1. Preliminaries

Throughout the paper, Σ denotes a first-order *signature*, i.e. a finite sequence of primitive relation and function symbols:

$$\Sigma=(r_1,\ldots,r_m;f_1,\ldots,f_n).$$

Every primitive function symbol f (relation symbol r) has a fixed nonnegative (positive) arity. Nullary function symbols are called *constants*. We always assume that Σ contains the equality symbol "$=$", and we also assume that Σ is always nontrivial, i.e. it contains at least one nonconstant function symbol. Nontrivial signatures are further classified into two categories. A signature Σ is said to be *poor* if the sum of all arities of the function and relation symbols in Σ is equal to 1. A nontrivial signature Σ is said to be *rich* if it is not poor.

It will be sometimes convenient to think of Σ as a 2-sorted signature, one of the sorts being always interpreted as the boolean sort. This allows us to think of relations as functions with boolean values. Naturally, we then include *true* and *false* as boolean constants in the signature.

A *Σ-structure* is an arbitrary set $\mathscr{A}$ together with an appropriate interpretation of the symbols in Σ, such that "$=$" is interpreted as equality. A *Σ-interpretation* is a pair $\mathscr{I}=(\mathscr{A},\boldsymbol{a})$, where $\mathscr{A}$ is a Σ-structure and $\boldsymbol{a}$ is a finite sequence of its elements which generate $\mathscr{A}$. If $\boldsymbol{a}$ is of length k, then $\mathscr{I}$ is called a *k-interpretation*. (This definition is motivated by the observation that all the values which occur in a computation of a program must belong to the (sub)structure generated by the input values.)

In the sequel we consider several classes of programs (programming languages), the syntax and semantics of which vary. We do not attempt to provide a most general definition of a program. Let us only agree at this moment that a program P over Σ will always have a fixed arity k (intuitively, the number of input arguments), and will define, for every Σ-structure $\mathscr{A}$, a partial function $P^{\mathscr{A}}:\mathscr{A}^k\to\mathscr{A}$, its semantics in $\mathscr{A}$. We say that P *converges* in $\mathscr{A}$ on input $\boldsymbol{a}\in\mathscr{A}^k$ if $P^{\mathscr{A}}(\boldsymbol{a})$ is defined; otherwise, it *diverges*. Two programs P_1 and P_2 are *equivalent* (written $P_1\equiv P_2$) iff, for every Σ-structure $\mathscr{A}$, $P_1^{\mathscr{A}}=P_2^{\mathscr{A}}$. We also say that P_1 and P_2 are *equivalent in an interpretation* $(\mathscr{A},\boldsymbol{a})$ iff $P_1^{\mathscr{A}}(\boldsymbol{a})=P_2^{\mathscr{A}}(\boldsymbol{a})$. A programming language $\mathscr{L}$ is *reducible* to another language $\mathscr{L}'$ (written $\mathscr{L}\leqslant\mathscr{L}'$) iff for every $P\in\mathscr{L}$ there exists a $P'\in\mathscr{L}'$ such that $P\equiv P'$. $\mathscr{L}$ is *equivalent* to $\mathscr{L}'$ ($\mathscr{L}\equiv\mathscr{L}'$) iff $\mathscr{L}\leqslant\mathscr{L}'$ and $\mathscr{L}'\leqslant\mathscr{L}$. The inequality $\mathscr{L}<\mathscr{L}'$ stands for $\mathscr{L}\leqslant\mathscr{L}'$ and $\mathscr{L}'\not\leqslant\mathscr{L}$.

The main issue of this paper is the investigation of functional languages to be introduced later. The analysis of the expressive power of these languages requires comparing them to certain classes of augmented flowchart programs. In this section we put together the main notions necessary for our purposes.

We assume that the reader is familiar with the notion of a flowchart, or a **while** program. Recall that a flowchart may be understood as a finite directed graph of nodes labelled by instructions. The instructions may take one of the following forms:

"$x := y$",

"$x := f(x_1, \ldots, x_n)$",

"$r(x_1, \ldots, x_n)?$".

Here $x, y, x_1, \ldots, x_n$ are variables, and f, r are, respectively, function and relation symbols in Σ.

Some of the variables in a program are chosen to be input variables, there is also one distinguished output variable. A flowchart has a single "start" node and possibly many "stop" nodes. If $\mathscr{A}$ is a Σ-structure and $\boldsymbol{a}$ is a vector of length equal to the number of input variables in a program P, then we may define in an obvious way a *computation* of P in $\mathscr{A}$ on the input $\boldsymbol{a}$ to be a sequence of *states*, each state being a pair of the form $(I, \boldsymbol{v})$, where I is an instruction (more formally, a node of the graph) and $\boldsymbol{v}$ is a valuation in A of all the variables in P. The noninput variables are assigned the first value in $\boldsymbol{a}$ by the initial valuation (auxiliary variables are initialized). The value of $P^{\mathscr{A}}(\boldsymbol{a})$ is then defined to be the value of the output variable in the final valuation, i.e. when a stop node is reached. With no loss of generality we may allow an additional sort of boolean variables to occur in a flowchart, thus, e.g. allowing tests of the form "$b?$", or assignments of the form "$b := r(\boldsymbol{x})$", where b is a boolean variable and r is a relation symbol.

We now extend the formalism of flowcharts by adding various features.

1.1. Counters

A *counter* is a special variable ranging over the set of nonnegative integers ω. A *flowchart with counters* may use a finite number of counters, occurring in instructions of the forms

"$c_1 := c_2$", "$c_1 := succ(c_2)$", "$c_1 := 0$", "$c_1 = c_2?$".

Counters are initialized to 0 at the beginning of a computation. Clearly, counters can simulate a Turing machine tape and, thus, e.g. the halting problem over finite interpretations is undecidable for such programs.

1.2. Pushdown stores

We use the abbreviation "pds" for the expression "pushdown store". A *level-1 pds* (or a 1-*pds*) over a set $\mathscr{A}$ is an arbitrary sequence $(a_1, \ldots, a_m)$ of elements of $\mathscr{A}$, with

$m \geqslant 0$. A *level-n pds* (or an *n-pds*), for $n \geqslant 2$, is a sequence $(s_1, \ldots, s_m)$ of $(n-1)$-pds's, where $m > 0$ (we assume that an "empty" n-pds contains an "empty" $(n-1)$-pds).

A *flowchart with an n-pds* is a flowchart allowing the following additional basic instructions:

(1-pds operations)	"$x := pop_1$",
	"$push_1(x)$",
(i-pds operations, for $1 < i \leqslant n$)	"pop_i",
	"$push_i$".

A *computation* of a flowchart with an n-pds in a Σ-structure $\mathscr{A}$ will be a sequence of *states*, each state consisting of a valuation of the individual variables, an n-pds over $\mathscr{A}$, and the next instruction to be executed. At the outset of a computation, the pds is empty. The meaning of the pds instructions is the following:

pop_n executed on $(s_1, \ldots, s_m)$ results in $(s_1, \ldots, s_{m-1})$;

$push_n$ executed on $(s_1, \ldots, s_m)$ results in $(s_1, \ldots, s_m, s_m)$.

The operations pop_i, $push_i$, for $1 < i < n$, are defined similarly, but apply to the "top" pds of level i. (The "top" $(n-1)$-pds of $s = (s_1, \ldots, s_m)$ is, of course, s_m, while the "top" i-pds of s, for $i < n-1$, is defined to be the "top" i-pds of s_m.) Operations on 1-pds have their usual meaning and are, of course, applied to the "top" 1-pds. Clearly, the notion of a flowchart with an n-pds is a direct generalization of the notion of a "flowchart with a pds", as considered for example in [36]. The language of flowcharts each with an n-pds is denoted as $\mathscr{L}_{n\text{-}pds}$, while $\mathscr{L}_{pds}$ is the union of $\mathscr{L}_{n\text{-}pds}$, for all $n \geqslant 1$. If k input variables and one output variable are specified for a $P \in \mathscr{L}_{pds}$, the operational semantics of P just outlined defines a (partial) function $P^{\mathscr{A}}$ in a given Σ-structure $\mathscr{A}$.

1.3. Arrays

A *k-ary array* is a nonsignature function symbol, say F, which is used in a program to denote a partial function

$$F : \mathscr{A}^k \to \mathscr{A},$$

where $\mathscr{A}$ is a structure. Such a function F is assumed to be initially empty, and the program may operate on it, using assignments of the forms

(*) "$F(x_1, \ldots, x_k) := x$" and "$x := F(x_1, \ldots, x_k)$".

Arrays of higher levels, informally, are arrays whose indices are arrays of lower levels, i.e. an array of level 2 denotes a function with function arguments, an array of level 3 stands for a function whose arguments are arrays of level 2 and so on. The instructions to operate on arrays of level $m > 1$ are

(**) "$\mathscr{F}(F_1, \ldots, F_k) := x$" and "$x := \mathscr{F}(F_1, \ldots, F_k)$",

where $\mathscr{F}$ is an array of level m, $F_1, \ldots, F_k$ are arrays of level $m-1$ and x is an individual variable. A *flowchart with arrays of level m* allows an arbitrary finite number of arrays of levels at most m and involves instructions of the form (*) and (**). For simplicity we assume here without loss of generality that each array in a given program has the same number of arguments k (we can thus avoid questions of type consistency). Now, a computation of such a program is again defined as an appropriate sequence of states, each state consisting of an instruction, a valuation of all individual variables and of a sequence of (possibly higher-order) functions assigned to the arrays. (An attempt to use an array cell of undefined value results in a diverging computation.) If the input variables and an output variable are specified, then a program $P \in \mathscr{L}_{array}$ defines in a Σ-structure $\mathscr{A}$ a function $P^{\mathscr{A}}$. The class of flowcharts with arrays is denoted as $\mathscr{L}_{array}$, while $\mathscr{L}_{n\text{-}array}$ denotes the subclass of $\mathscr{L}_{array}$ consisting of all programs P such that the maximum level of arrays occurring in P is n. Of course, $\mathscr{L}_{1\text{-}array}$ is exactly the class FDA of "flowcharts with arrays" considered in [36].

For a more detailed exposition of programs with stacks and arrays the reader is referred to [35]. The additional programming features introduced above may be combined: in particular, one may consider flowcharts with pushdown stores and counters. In the literature it is usually agreed that flowcharts with 1-pds's and counters form a "universal class" of programs, i.e. that they correspond to the most general notion of a computing device effective relative to an underlying algebraic structure, see e.g. [1, 16, 21, 26].

In order to classify the expressive power of programs, each augmented with a push-down store and arrays, one must first note that, by [37], a program in $\mathscr{L}_{1\text{-}pds}$ is able to enumerate all elements in any given Σ-interpretation. Thus, in particular, in infinite interpretations $\mathscr{L}_{1\text{-}pds}$-programs can simulate counters. This means that $\mathscr{L}_{1\text{-}pds}$ is a universal class over all infinite interpretations. The same holds for $\mathscr{L}_{1\text{-}array}$, by [36]. Languages of this property are called *semiuniversal*. Obviously, then the analysis of their computational power naturally leads to investigating behaviour of programs over finite interpretations.

We briefly recall here the approach of [36] and [35], which allows us to compare semiuniversal languages over finite interpretations. Let us assume that there is a given coding c which assigns to each finite interpretation $\mathscr{I} = (\mathscr{A}, \boldsymbol{a})$, a string $c(\mathscr{I}) \in \{0,1\}^*$, such that for k-interpretations $\mathscr{I}$ and $\mathscr{I}'$, $c(\mathscr{I}) = c(\mathscr{I}')$ iff $\mathscr{I}$ and $\mathscr{I}'$ are isomorphic. The *spectrum* of a language $\mathscr{L}$ is the set

$$sp(\mathscr{L}) = \{sp_k(P) \mid k \geqslant 0 \text{ and } P \text{ is a } k\text{-ary program in } \mathscr{L}\},$$

where $sp_k(P)$ is the spectrum of P, i.e. the set of all codes of finite k-interpretations $\mathscr{I}$ such that P converges in $\mathscr{I}$.

Let now $St_k = \{c(\mathscr{I}) \mid \mathscr{I} \text{ is a } k\text{-interpretation}\}$ and let $\mathscr{C}$ be a complexity class. We say that the *spectral complexity of* $\mathscr{L}$ *is in* $\mathscr{C}$ if $sp(\mathscr{L}) \subseteq \mathscr{C}$. The spectral complexity of $\mathscr{L}$ is *hard* in $\mathscr{C}$ iff for every k and every language $C \in \mathscr{C}$ such that $C \subseteq St_k$, there exists a program $P \in \mathscr{L}$ with $C = sp_k(P)$. Finally, the spectral complexity of $\mathscr{L}$ is *equal* to $\mathscr{C}$ if both conditions hold.

Let us notice that the notion of spectral complexity of a given programming language depends on a coding function c. The choice of c may also depend on the signature Σ. The essential difference between rich and poor signatures, whose impact will be seen in the next theorem, is that the coding function for rich signatures can be defined so that the length of a code of $\mathscr{I}$ is polynomial in the cardinality of the universe of $\mathscr{I}$, while for poor signatures this length can be made logarithmic.

Let $exp_0(n)=n$ and $exp_{k+1}(n)=2^{exp_k(n)}$. We have the following characterization of the spectral complexities of our languages.

1.1. Theorem (Tiuryn [35, Theorem 4.1.8]). (i) *For every rich signature Σ, there exists a coding function c such that for every $k \geqslant 1$, the spectral complexity of $\mathscr{L}_{k\text{-}pds}$ is equal to $DTIME(exp_k(\mathscr{O}(\log n)))$ and the spectral complexity of $\mathscr{L}_{k\text{-}array}$ is equal to $DSPACE(exp_k(\mathscr{O}(\log n)))$.*

(ii) *For every poor signature Σ, there exists a coding function c such that for every $k \geqslant 1$, the spectral complexity of $\mathscr{L}_{k\text{-}pds}$ is equal to $DTIME(exp_k(\mathscr{O}(n)))$ and the spectral complexity of $\mathscr{L}_{k\text{-}array}$ is equal to $DSPACE(exp_k(\mathscr{O}(n)))$.*

The proof of the theorem involves the methods developed in [36], where a special case ($k=1$) was considered. For $k>1$, the analysis for $\mathscr{L}_{k\text{-}pds}$ uses a result of [25] which provides a complexity-theoretic characterization of formal languages recognized by "auxiliary pushdown automata of level k". The latter is a generalization of the notion of an "APDA" introduced by Cook (see [14]). The coding functions mentioned in the above theorem are constructed in [35] explicitly. We shall refer to them in the proof of Theorem 3.7.

It also follows from the results of [35, Theorem 4.1.5] that the languages $\mathscr{L}_{k\text{-}pds}$ and $\mathscr{L}_{k\text{-}array}$ form infinite hierarchies

$$\mathscr{L}_{1\text{-}pds} < \mathscr{L}_{2\text{-}pds} < \cdots,$$

$$\mathscr{L}_{1\text{-}array} < \mathscr{L}_{2\text{-}array} < \cdots,$$

which are connected by the inequalities

$$\mathscr{L}_{k\text{-}pds} \leqslant \mathscr{L}_{k\text{-}array} \leqslant \mathscr{L}_{k+1\text{-}pds}$$

for every $k \geqslant 1$ and for every nontrivial signature. Another important property of the languages $\mathscr{L}_{k\text{-}pds}$ and $\mathscr{L}_{k\text{-}array}$ is that they are divergence-closed, i.e. for an arbitrary n and arbitrary n-ary program P in $\mathscr{L}_{k\text{-}pds}$ ($\mathscr{L}_{k\text{-}array}$) there exists another program P' in $\mathscr{L}_{k\text{-}pds}$ ($\mathscr{L}_{k\text{-}array}$), such that $sp_n(P')=St_n-sp_n(P)$, i.e. the spectrum of P' is the complement of the spectrum of P.

2. The untyped language

Our goal is to study typed languages. However, the syntax and semantics of our languages has a common base, which is the untyped lambda calculus extended with

certain additional constructs. The typed languages are then defined by introducing type-inference systems, so that only typable expressions are considered. The semantics of all our languages is the same as the semantics of the untyped language. In this section we introduce our untyped language, describe its semantics and state several useful facts.

2.1. Syntax

The syntax is defined with respect to a first-order signature Σ, and a Σ-structure $\mathscr{A}$. The set of *identifiers* consists of

- the constants *true* and *false*;
- an infinite set of *variables*, called also *object variables*, to distinguish from "type variables", to be introduced later;
- all symbols of the signature Σ, including equality; for uniformity we assume that there are two different symbols for equality: between elements of $\mathscr{A}$ and between boolean values; in prefix notation equality is denoted by $eq^{\mathbf{0}}$ and $eq^{\mathbf{Bool}}$;
- additional *algebraic constants* denoting all elements of $\mathscr{A}$.

All identifiers, except variables, are called *constants*. The set of all *generalized λ-terms* (or simply "terms") is defined by the following BNF-grammar.

$$\begin{aligned}\langle term\rangle ::= {} & \langle identifier\rangle \mid (\langle term\rangle\langle term\rangle) \mid (\lambda\langle variable\rangle.\langle term\rangle)\\ & \mid (\mathbf{fix}\,\langle variable\rangle.\langle term\rangle) \mid (\mathbf{let}\,\langle variable\rangle = \langle term\rangle\,\mathbf{in}\,\langle term\rangle)\\ & \mid (\mathbf{if}\,\langle term\rangle\,\mathbf{then}\,\langle term\rangle\,\mathbf{else}\,\langle term\rangle).\end{aligned}$$

We follow the usual convention that MNP stands for $((MN)P)$. Generalized λ-terms with no occurrence of algebraic constants are called just *λ-terms*. (Thus, the notion of a λ-term does not depend on $\mathscr{A}$.) Further, we define (*generalized*) *applicative terms* to be (generalized) λ-terms with no occurrence of λ, **fix** or **let**. The equality on terms (as syntactic objects) is denoted by "$\equiv$".

The set $FV(M)$ of all free variables in a term M is defined as follows.

$$FV(x)=\{x\}, \quad \text{if } x \text{ is a variable;}$$

$$FV(c)=\emptyset, \quad \text{if } c \text{ is a constant;}$$

$$FV(MN)=FV(M)\cup FV(N);$$

$$FV(\lambda x.\,M)=FV(M)-\{x\};$$

$$FV(\mathbf{fix}\,x.\,M)=FV(M)-\{x\};$$

$$FV(\mathbf{let}\,x=M\,\mathbf{in}\,N)=FV(M)\cup(FV(N)-\{x\});$$

$$FV(\mathbf{if}\,B\,\mathbf{then}\,N\,\mathbf{else}\,M)=FV(B)\cup FV(N)\cup FV(M).$$

Note that the scope of the **let**-binding in $(\mathbf{let}\,x=M\,\mathbf{in}\,N)$ is only N. The notion of α-conversion is defined in a standard way. Terms which are α-convertible are assumed

equal. A term, or a collection of terms, is called *α-correct* iff no free variable is bound elsewhere and no variable is bound twice. The notation $M[N/x]$ stands for the term obtained from M by substituting N for all free occurrences of x. We also use the notation $M[N_1/x_1, \ldots, N_k/x_k]$ for simultaneous substitution. Sometimes a vector of variables will be denoted by $\boldsymbol{x}$, and similarly for terms. Thus, we will use notations such as: $\lambda \boldsymbol{x}.\, M$ or $M[\boldsymbol{N}/\boldsymbol{x}]$.

A *function definition* is an equation of the form

$$"F = T",$$

where T is a λ-term and F is a variable. A function definition is *simple* iff it takes the form

$$"F = \lambda \boldsymbol{x}.\, M",$$

where M is an applicative term. Whenever we consider sets of function definitions we always assume that, for each F, there is at most one definition with F at the left-hand side.

2.2. *Semantics*

An operational semantics for generalized λ-terms is defined with respect to a Σ-structure $\mathscr{A}$ and a set of function definitions D (possibly, empty) by means of reduction rules. We use the notation $\mathscr{A}, D \models M_1 \to M_2$ for one-step reduction and $\mathscr{A}, D \models M_1 \twoheadrightarrow M_2$ for many-step reduction. We skip one or both of $\mathscr{A}$ and D in the notation if no reduction depending on them is involved, or if $\mathscr{A}, D$ are known from the context. The reduction rules are as follows:

β-reduction:	$\models (\lambda x.\, M)N \to M[N/x]$;
δ(algebraic)-reduction:	$\mathscr{A} \models fa_1 \ldots a_n \to a$ if f is a k-ary function (relation) symbol in Σ, and $f(a_1, \ldots, a_n) = a$ holds in $\mathscr{A}$;
if-reductions:	$\models$ (**if** *true* **then** N **else** $M) \to N$, and $\models$ (**if** *false* **then** N **else** $M) \to M$;
let-reduction:	$\models$ (**let** $x = N$ **in** $M) \to M[N/x]$;
fix-reduction:	$\models$ **fix** $x.\, M \to M$[**fix** $x.\, M/x$];
D-reductions:	$D \models F \to T$, for all "$F = T$" in D.

2.1. Proposition. *The above reduction system has the Church–Rosser property and the standarization property, i.e. if a term M has a normal form, then the normal form may be obtained from M by reducing at each step the leftmost redex in a term.* (*Leftmost reductions are denoted by* $\to_L$ *and* $\twoheadrightarrow_L$.)

Proof. This proposition is a consequence of Theorem 3.11 in Chapter II of [24]. In our case we have a "regular combinatory reduction system" (in the terminology of [24]). In [24] the Church–Rosser property is proved for untyped reduction systems, which immediately implies that the property also holds for typed versions of the same systems. □

If D is a set of function definitions, then a term M is called *D-closed* if all free variables of M are defined in D. A *semiprogram* is a pair (D, M), where D is a set of function definitions and M is a λ-term, such that the right-hand sides of the definitions and M are D-closed. We usually assume α-correctness of M and (the right-hand sides of) the definitions in D. A semiprogram is *simple* iff all definitions are simple and M is an applicative term. Two semiprograms (D, M) and (D', M') are *reduction-equivalent* iff, for all k and all $a_1, \ldots, a_k, a \in \mathscr{A} \cup \{true, false\}$:

$$\mathscr{A}, D \models M a_1 \ldots a_k \twoheadrightarrow_{\mathrm{L}} a \text{ iff } \mathscr{A}, D' \models M' a_1 \ldots a_k \twoheadrightarrow_{\mathrm{L}} a.$$

It is more convenient to deal with simple semiprograms, or semiprograms containing no definitions (i.e. just λ-terms). We are now going to describe transformations on semiprograms which turn an arbitrary semiprogram into a simple one or into a single λ-term.

2.3. *Transformation I*

Give an arbitrary semiprogram (D, M), we construct a simple semiprogram as follows. First we eliminate all occurrences of **fix** and **let** in M and in the definitions, starting from the outermost **let**- or **fix**-redexes and ending with the innermost ones. Second, we eliminate all λ-bindings, starting from the innermost and ending with the outermost (except the outermost λ's in definitions). We describe now the particular steps.

Step 1. Let C be an outermost **let**- or **fix**-redex in M.

Case (*a*): $C \equiv \mathbf{fix}\, x.\, P$. Let $\boldsymbol{y}$ be all free variables of C except those defined in D, and let X be a new variable. Replace C by $X\boldsymbol{y}$ and add a new definition

"$X = \lambda \boldsymbol{y}.\, P[X\boldsymbol{y}/x]$".

Case (*b*): $C \equiv \mathbf{let}\, x = P\, \mathbf{in}\, Q$. We describe two strategies to eliminate an outermost **let**. Both are equally good with respect to reduction-equivalence of untyped semiprograms. However, when we later consider our transformation for typed expressions, we will have to ensure that it does not change typability, and we will choose one of the two methods depending on the typing discipline.

Method 1: Again, let $\boldsymbol{y}$ be all free variables of C except those defined in D. Let $x^{(1)}, \ldots, x^{(n)}$ be all the occurrences of x in Q (if $n = 0$, then we proceed as if $n = 1$). Take new variables $X_1, \ldots, X_n$ and add definitions

$$X_j = \lambda \boldsymbol{y}.\, P,$$

for all $j=1,\ldots,n$. Replace C by $Q[X_j\boldsymbol{y}/x^{(j)}]_{j=1}^{n}$, i.e. by the term obtained from Q by replacing each $x^{(j)}$ with $X_j\boldsymbol{y}$.

Method 2: This differs from Method 1 in that we do not distinguish between different occurrences of x. We add one definition

$$X=\lambda\boldsymbol{y}.\,P,$$

and replace C with $Q[X\boldsymbol{y}/x]$.

Step 2. Now we assume that our semiprogram does not involve occurrences of **let** or **fix**. Choose an innermost subterm of the form

$$C\equiv\lambda x.\,P,$$

and let $\boldsymbol{y}$ be all free variables of C except those defined in D. Let F be a new variable. Replace C by $F\boldsymbol{y}$ and add a new definition

$$F=\lambda\boldsymbol{y}x.\,P.$$

2.4. Transformation II

Given an arbitrary semiprogram (D, M), we construct a λ-term M_D (a semiprogram of the form $(\emptyset, M_D)$). We eliminate one definition at one step. Choose a definition "$F=T$" in D and take a new variable F'. Let T' be $T[F'/F]$. We define

$$M'\equiv\textbf{let}\ F=(\textbf{fix}\ F'.\,T')\ \textbf{in}\ M,$$

$$D'=\{\text{“}G=(\textbf{let}\ F=(\textbf{fix}\ F'.\,T')\ \textbf{in}\ P)\text{”}\mid\text{“}G=P\text{”}\in D,\ G\not\equiv F\}.$$

Clearly, (D', M') has less definitions, and we can proceed this way until the set of definitions is empty.

2.2. Lemma. *Let (D, M_0) be a semiprogram and let (D', M_0') be obtained from (D, M_0) by Transformation I. Then (D, M_0) and (D', M_0') are reduction-equivalent.*

Proof. Consider one step in the transformation, consisting of a replacement of a term C by another term C', and adding new definition(s), as described above. Here C is either (**let** $x=P$ **in** Q) or (**fix** $x.\,P$) or ($\lambda x.\,P$). We show that a semiprogram obtained from (D, M_0) by such a single step is reduction-equivalent to (D, M_0). Assume that (D, M_0) is α-correct and that $D=\{\text{“}F_i=M_i\text{”}\mid i=1,\ldots,N\}$ and that $\boldsymbol{y}=(y_1,\ldots,y_m)$ are all variables free in C except for the F_i's. We define a set of *good* terms as follows:

- a constant or a variable different from x is good;
- if B, N, M are good and y is a variable different from x, then (MN), (**if** B **then** M **else** N) ($\lambda y.\,M$), (**let** $y=M$ **in** N), (**fix** $y.\,M$) are good (the latter two not applicable to step 2 in Transformation I);

– if $I \subseteq \{1,\ldots,m\}$ and N_i are good and D-closed, for $i \in I$, then $C[\boldsymbol{N}/\boldsymbol{y}]_I$ is good, where $C[\boldsymbol{N}/\boldsymbol{y}]_I$ denotes the result of a simultaneous substitution of N_i for y_i, for all $i \in I$.
Terms of the latter form are called *instances* of C.

For good M, we define a term T_M as follows. Let M be an instance of C, say $C[\boldsymbol{N}/\boldsymbol{y}]_I$, and let $\boldsymbol{T}$ be a sequence $T_1,\ldots,T_m$ such that $T_i \equiv T_{N_i}$ for $i \in I$, and $T_i \equiv y_i$ otherwise. Now T_M is

$X\boldsymbol{T}$	for Step 1(a);
$Q[X_1\boldsymbol{T}/x^{(1)}\ldots X_n\boldsymbol{T}/x^{(n)}]$	for Step 1(b), Method 1;
$Q[X\boldsymbol{T}/x]$	for Step 1(b), Method 2;
$F\boldsymbol{T}$	for Step 2.

In the remaining cases

$T_m \equiv M$, if M is a constant or a variable;

$T_{MN} \equiv T_M T_N$,

$T_{\textbf{if } B \textbf{ then } M \textbf{ else } N} \equiv \textbf{if } T_B \textbf{ then } T_M \textbf{ else } T_N$,

$T_{\lambda y.\, M} \equiv \lambda y.\, T_M$,

$T_{\textbf{let } y = M \textbf{ in } N} \equiv \textbf{let } y = T_M \textbf{ in } T_N$,

$T_{\textbf{fix } y.\, M} \equiv \textbf{fix } y.\, T_M$.

Note that $T_C \equiv C'$ and that $T_M \equiv M$, if no instance of C occurs in M. For all good M, we have $FV(T_M) \subseteq M$. The following two claims may now be easily proved by induction.

Claim 1. *If M, N are good, $x \not\equiv y$, and N is D-closed, then $M[N \mid y]$ is good and $T_{M[N/y]} \equiv T_M[T_N/y]$.*

Claim 2. *Let M be a good and D-closed term, not of the form $\lambda y.\, N$. If $D \models M \to_{\mathrm{L}} M'$ then M' is good and $D' \models T_M \twoheadrightarrow_{\mathrm{L}} T'_M$. Moreover, the latter reduction takes at least one step, except for the case when the leftmost redex is an instance of C in Step* 1(b).

To complete the proof of our lemma it suffices to show that

$$\mathscr{A}, D \models M_0\boldsymbol{a} \twoheadrightarrow_{\mathrm{L}} a \text{ iff } \mathscr{A}, D' \models T_{M_0}\boldsymbol{a} \twoheadrightarrow_{\mathrm{L}} a$$

for all $\boldsymbol{a}$ and a in $\mathscr{A}$ since $T_{M_0} \equiv M'_0$. The "only if" part follows immediately from Claim 2. For the "if" part, consider the leftmost derivation starting from $M_0\boldsymbol{a}$. If it ends with a normal form different from a, or it leads to an abstraction, then the same must hold for the derivation starting from $T_{M_0}\boldsymbol{a}$. If $M_0\boldsymbol{a}$ has no normal form, i.e. the derivation is infinite, then the other derivation must also be infinite: a step of the form $M_1 \to_{\mathrm{L}} M_2$ with $T_{M_1} \equiv T_{M_2}$ would otherwise occur infinitely many times in a row – this is impossible since there are only finitely many **let**'s in M_0. □

2.3. Lemma. *Let (D, M) be a semiprogram and let $(\emptyset, M_D)$ be obtained from (D, M) by Transformation II. Then (D, M) and $(\emptyset, M_D)$ are reduction-equivalent.*

Proof. Again we show the correctness of a single step. Using the notation of Transformation II, let N^* be $N[\mathbf{fix}\, F'.\, T'/F]$, for any N. We show by an easy induction the following claim.

Claim. *If $\mathscr{A}, D \models N \to_L N_1$ then $\mathscr{A}, D' \models N^* \twoheadrightarrow_L N_1^*$ (in at least one step). It remains to show that*

$$\mathscr{A}, D \models M\boldsymbol{a} \twoheadrightarrow_L a \text{ iff } \mathscr{A}, D' \models M'\boldsymbol{a} \twoheadrightarrow_L a,$$

which is established as in the previous proof. □

3. Finitely typed recursive procedures

The set of all *finite* (*monomorphic*) *types* is the least set containing **0** (the ground type of individuals) and **Bool** (the ground type of truth values), and containing $(\sigma \to \tau)$ whenever it contains σ and τ. We use the abbreviation $\sigma_1 \to \cdots \to \sigma_n \to \tau$ for $(\sigma_1 \to (\cdots (\sigma_n \to \tau) \cdots))$. If, for all i, $\sigma_i = \sigma$, then we also write $\sigma^n \to \tau$, identifying $\sigma^0 \to \tau$ with τ. The *level* of a type σ is

$$\ell(\sigma) = \begin{cases} 0 & \text{if } \sigma = \mathbf{0} \text{ or } \mathbf{Bool}; \\ max\{\ell(\tau_1)+1, \ell(\tau_2)\} & \text{if } \sigma = (\tau_1 \to \tau_2). \end{cases}$$

Every type σ may be expanded into the form $\sigma = \sigma_1 \to \cdots \to \sigma_n \to \mathbf{a}$, where $\mathbf{a} \in \{\mathbf{0}, \mathbf{Bool}\}$, and an equivalent definition of the level of σ is

$$\ell(\sigma) = 1 + max\{\ell(\sigma_1), \ldots, \ell(\sigma_n)\}.$$

For the present section we restrict ourselves to monomorphic types; thus, "type" means here always "monomorphic type". We are going to define a programming language FL of finitely typed programs over a given signature Σ. The syntax of the language is determined by a type inference system for generalized λ-terms (with respect to a Σ-structure $\mathscr{A}$), i.e. a legal expression of the language is the one which can be assigned a type.

In order to state a type assertion system, let us define an *environment* to be a set E of *type assumptions* of the form $(x : \sigma)$, where x is a variable and σ is a type, such that if $(x : \sigma), (x : \sigma') \in E$, then $\sigma = \sigma'$. Thus, one can think of an environment as a finite partial function from variables into types. If E is an environment, then $E(x : \sigma)$ is an environment such that

$$E(x : \sigma)(y) = \begin{cases} \sigma & \text{if } x \equiv y; \\ E(y) & \text{if } x \not\equiv y. \end{cases}$$

First we assign types to constant symbols. The symbols *true* and *false* are assumed to be of type **Bool**, algebraic constants are of type **0** and, finally, function and relation symbols in Σ of arity k are of types $\mathbf{0}^k \to \mathbf{0}$ and $\mathbf{0}^k \to \mathbf{Bool}$, respectively. In particular, $eq^{\mathbf{0}}$ is of type $\mathbf{0}^2 \to \mathbf{Bool}$, and $eq^{\mathbf{Bool}}$ of type $\mathbf{Bool}^2 \to \mathbf{Bool}$.

The type assertion system for FL consists of the following rules (here τ, τ' are monomorphic types). We read "$E \vdash M : \tau$" as "M has the type τ in the environment E". If $E = \emptyset$, then we write simply "$\vdash M : \tau$". (In the sequel we sometimes write "$E \vdash_{FL} M : \tau$" for "$E \vdash M : \tau$" in order to stress the typability in this particular system.)

(CONST) $E \vdash c : \tau$, where τ is the type of a constant c.

(VAR) $E \vdash x : \tau$, if $(x : \tau)$ is in E.

(APP) $$\frac{E \vdash M : \tau \to \tau',\ E \vdash N : \tau}{E \vdash (MN) : \tau'}.$$

(COND) $$\frac{E \vdash B : \mathbf{Bool},\ E \vdash M : \tau,\ E \vdash N : \tau}{E \vdash (\mathbf{if}\ B\ \mathbf{then}\ M\ \mathbf{else}\ N) : \tau}.$$

(ABS) $$\frac{E(x : \tau) \vdash M : \tau'}{E \vdash (\lambda x.\, M) : \tau \to \tau'}.$$

(FIX) $$\frac{E(x : \tau) \vdash M : \tau}{E \vdash (\mathbf{fix}\ x.\, M) : \tau}.$$

(LET) $$\frac{E \vdash N : \tau,\ E(x : \tau) \vdash M : \tau'}{E \vdash (\mathbf{let}\ x = N\ \mathbf{in}\ M) : \tau'}.$$

We classified types with respect to their levels. The above type assertion system may easily be modified so that the types used in the derivations are of a bounded level. Let $n > 0$ be a fixed bound. We write $E \vdash_n M : \tau$ if $E \vdash M : \tau$ can be derived in a subsystem obtained by

- adding a restriction that $level(\tau)$, $level(\tau') \leqslant n$ in all the rules;
- adding a restriction that $level(\tau) \leqslant n - 1$ in the rule ABS.

A *program* in FL is a closed λ-term M such that

$$\vdash_{FL} M : \mathbf{0}^k \to \mathbf{0}$$

for some k. Such a program M defines in every Σ-structure $\mathscr{A}$ a k-ary partial function

$$M^{\mathscr{A}} : \mathscr{A}^k \to \mathscr{A}$$

defined by

$$M^{\mathscr{A}}(a_1, \ldots, a_k) = a \ \text{ iff } \ \mathscr{A} \models Ma_1 \ldots a_k \twoheadrightarrow a.$$

The class of all FL-programs is naturally stratified by the level of types used. We say that a program M is of *level* n if $\vdash_n M : \mathbf{0}^k \to \mathbf{0}$ and n is the least with this property. The notation FL_n stands for the sublanguage of all programs of level at most n.

A set of definitions D is *typable* in FL_n iff there exists an environment E such that

$$E \vdash_n T : E(F), \text{ for all } "F = T" \text{ in } D.$$

If (D, M) is a semiprogram, and in addition

$$E \vdash_n M : \tau, \text{ for some } \tau,$$

then we also say that (D, M) is typable and that it is of level at most n and has the type τ in E. We write $E \vdash_n (D, M) : \tau$. A semiprogram (a set of definitions) is *typable* in FL iff it is typable in FL_n, for some n. Finally, a *recursive procedure* in FL (FL_n) is a simple semiprogram (D, M) of type $\mathbf{0}^k \to \mathbf{0}$.

Let us observe that our reduction rules are consistent with the type assertion systems, and that we can equally well consider programs and recursive procedures.

3.1. Lemma. *If $E \vdash_n M : \tau$ and $E \vdash_n T : E(F)$ for all "$F = T$" in a set of definitions D, then $\mathscr{A}$, $D \models M \twoheadrightarrow M'$ implies $E \vdash_n M' : \tau$. In particular, if (D, M) is a recursive procedure in FL_n of type $\mathbf{0}^k \to \mathbf{0}$ and $M\boldsymbol{a}$ has a normal form a, then $a \in \mathscr{A}$.*

Proof. See Lemma A.4. □

3.2. Lemma. *For each FL_n-program there exists an equivalent recursive procedure in FL_n, and vice versa.*

Proof. See Lemmas A.5 and A.6. □

3.3. Example. Assume a signature Σ of two unary function symbols f and g and one unary relation symbol r. Let D consist of the following function definitions:

$$\mathscr{F} = \lambda xy.\, \mathscr{G} x g y;$$

$$\mathscr{G} = \lambda x F y.\ \textbf{if } rx \textbf{ then } gy \textbf{ else } \mathscr{G}(fx)(\mathscr{H}F)(Fy);$$

$$\mathscr{H} = \lambda F x.\, F(Fx).$$

One can easily check that $P = (D, M)$ is a recursive procedure in FL_2, of type $\mathbf{0}^2 \to \mathbf{0}$ in the following environment:

$$\mathscr{F} : \mathbf{0}^2 \to \mathbf{0},$$

$$\mathscr{G} : \mathbf{0} \to (\mathbf{0} \to \mathbf{0}) \to \mathbf{0} \to \mathbf{0},$$

$$\mathscr{H} : (\mathbf{0} \to \mathbf{0}) \to \mathbf{0} \to \mathbf{0}.$$

Assume now that, for some interpretation $(\mathscr{A}, (a, b))$, we have

(*) $\quad r(f^n a) = \mathit{true} \quad \text{and} \quad r(f^i a) = \mathit{false}, \qquad \text{for all } i < n.$

Then one can verify that $\mathscr{A}$, $D \models \mathscr{F} ab \twoheadrightarrow g^{2^n} b$.

We address here the issue of the expressive power of FL_n-programs (or, equivalently, recursive procedures of level n) for arbitrary n. We start with a lemma which guarantees that all the languages FL_n are semiuniversal.

3.4. Lemma (Urzyczyn [37]). *For each $k>0$ there exists a level-1 finitely typed recursive procedure $P_{Next}=(D_{Next}, Next)$ in FL_1 of type $\mathbf{0}^{k+1}\to\mathbf{0}$ such that Next is a function variable, and for every k-interpretation $(\mathscr{A}, \boldsymbol{a})$ with $\boldsymbol{a}=(a_1,\ldots,a_k)$, the function*

$$P^{\mathscr{A}}_{Next}: \mathscr{A}^{k+1}\to\mathscr{A}$$

is total, and if $b_i\in\mathscr{A}$ denotes the value of $(Next\ a_1\ldots a_k)^i a_1$, then

(a) $\mathscr{A}=\{b_i \mid i\in\omega\}$;

(b) *all b_i, for $i<|\mathscr{A}|$ are pairwise different (all b_i are different if $\mathscr{A}$ is infinite), and if $j\geqslant i=|\mathscr{A}|$, then $b_j=b_{i-1}$.*

First we raise the question of a "lower bound" for the computational power of FL_n. For this we compare it with $\mathscr{L}_{k\text{-}pds}$ – the language of flowcharts, each augmented with a level-k pushdown store.

3.5. Theorem (Kfoury et al. [17]). *For all $k\geqslant 1$, it holds that $\mathscr{L}_{k\text{-}pds}\leqslant FL_k$.*

In [17], we have established the following "upper bound":

3.6. Theorem (Kfoury et al. [17]). *For every $k\geqslant 1$, it holds that $FL_k\leqslant\mathscr{L}_{k\text{-}array}$.*

As a consequence, we obtain an infinite, "almost strict" hierarchy, i.e. we conclude that $FL_k<FL_{k+2}$ for all k.

We do not reproduce here the original proofs of Theorems 3.5 and 3.6. In particular, the proof of Theorem 3.5 was quite complicated and obtained by a detour into imperative languages, based on the results of [22, 23]. After the publication of our proofs for Theorems 3.5 and 3.6 in [17], Goerdt [12] was able to essentially simplify the proof of Theorem 3.5, and to refine that of Theorem 3.6 to obtain a stronger result. Namely, we have the following theorem.

3.7. Theorem (Goerdt [12]). *For all $k\geqslant 1$, the spectral complexity of the language FL_k is equal to $DTIME(exp_k(\mathcal{O}(\log n)))$.*

Corollary 3.8 follows from this theorem.

3.8. Corollary. *For all $k\geqslant 1$, the languages FL_k and $\mathscr{L}_{k\text{-}pds}$ are equivalent.*

Proof. Let P be an FL_k-program of arity $m\geqslant 1$. Since $\mathscr{L}_{k\text{-}pds}$ is semiuniversal, there exists a program Q_{inf}, which is equivalent to P over all infinite interpretations (this

program simulates an appropriate flowchart with level-1 pushdown store and counters, so that the function *Next* is used as the successor). We can also construct a program Q_{fin} in $\mathscr{L}_{k\text{-}pds}$, equivalent to P over all finite interpretations. Indeed, let P' be an FL_k-program of arity $m+1$ such that, for any given interpretation $(\mathscr{A}, \boldsymbol{a})$ and any $b \in \mathscr{A}$, we have

$$P'\boldsymbol{a}b \text{ converges in } \mathscr{A} \text{ iff } \mathscr{A} \models P\boldsymbol{a} \twoheadrightarrow b.$$

The spectrum of P' is in $DTIME(exp_k(\mathcal{O}(\log n)))$; thus, there exists a program Q' in $\mathscr{L}_{k\text{-}pds}$ which converges on a finite interpretation if and only if so does P'. Using the fact that $\mathscr{L}_{k\text{-}pds}$ is divergence-closed and semiuniversal (see [35]), one may now construct Q_{fin} so that, for an input $\boldsymbol{a}$, it will generate all possible values of b and check whether Q' halts on $\boldsymbol{a}$ and b.

It remains to "put together" the programs Q_{fin} and Q_{inf} to construct a program Q, equivalent to P over all interpretations. This is done using a standard trick: Q will behave as if the interpretation were infinite, as long as the simulation of counters with the help of *Next* is correct, i.e. until it is discovered that *Next* $\boldsymbol{a}$ a evaluates to a, for some a. If this happens, the program runs Q_{fin} on the original input.

Now, assume that P is a program in $\mathscr{L}_{k\text{-}pds}$. To construct a program Q in FL_k, one proceeds in a similar way as above. Note that we do not need to provide a separate proof that FL_k is divergence-closed, as we need this property only for a program whose spectrum is equal to a spectrum of an $\mathscr{L}_{k\text{-}pds}$-program, and we know that $\mathscr{L}_{k\text{-}pds}$ is divergence-closed. The details are left to the reader. (The construction of a program which unifies the finite and infinite cases is now slightly more complex, because one has to implement the construction for functional programs.) □

This generalizes the known result of [1] and [3], that the power of ordinary recursion is equivalent to that of a pushdown store (of level 1). As a further consequence, we obtain the following theorem.

3.9. Theorem. *The higher-level functional programming languages FL_n form a strict hierarchy of the form $FL_1 < FL_2 < \cdots$.*

For another proof of the strictness of the hierarchy (by a direct diagonalization) see [11].

It also follows from Corollary 3.8 that the union FL of the hierarchy is equivalent to both $\mathscr{L}_{array}$ and $\mathscr{L}_{pds}$ and, thus, its spectral complexity is equal to the class of all elementarily recursive languages.

4. ML-polymorphic recursive procedures

We are now going to extend our language by allowing programs typable with the help of universally polymorphic types. In order to define the polymorphic types, let us assume an infinite set of *type variables*, (denoted by $\alpha, \beta, \ldots$, possibly with subscripts).

We define *open types* as follows:

- The type constants **0** and **Bool** are open types.
- The type variables are open types.
- If τ and τ' are open types, then $(\tau \to \tau')$ is an open type.

The set of all *types* is the smallest set satisfying the following conditions.

- An open type is a type.
- If σ is a type and α is a type variable, then $\forall\alpha.\sigma$ is a type.

Thus, only outermost quantifiers may occur in types: each type σ is of the form $\forall \boldsymbol{\alpha}.\tau$, where τ is open. We use the notation $\tau = body(\sigma)$. Types which differ only in the order of outermost quantifiers are assumed equal. The notion of a free type variable in a type and that of a substitution $\sigma[\tau/\alpha]$ of an open type τ for all free occurrences of α in σ are defined in the usual way. Let $FV(\sigma)$ stand for the set of type variables free in σ. We also use the notation $\sigma[\tau_1/\alpha_1, \ldots, \tau_m/\alpha_m]$ for simultaneous substitutions (in shortened form: $\sigma[\boldsymbol{\tau}/\boldsymbol{\alpha}]$). A type with no free variables is called *closed*. Note that all monomorphic types are types (both closed and open). A type σ' is called an *instance* of $\sigma = \forall\alpha_1 \ldots \alpha_m.\tau$ (denoted as $\sigma \preceq \sigma'$) if σ' is of the form $\forall\beta_1 \ldots \beta_n.\tau'$, where $\tau' = \tau[\tau_1/\alpha_1, \ldots, \tau_m/\alpha_m]$, for some open types $\tau_1, \ldots, \tau_m$ and $\beta_1, \ldots, \beta_n \notin FV(\sigma)$. (Note that, if $\boldsymbol{\alpha}$ are not free in $\boldsymbol{\tau}$, then the substitution may be considered sequential.) If σ is a type, then by $\forall.\sigma$ we denote the type $\forall\boldsymbol{\alpha}.\sigma$, where $\boldsymbol{\alpha}$ are all the variables free in σ. If types σ and σ' are identical except for renaming of their bound variables, then we write $\sigma =_\alpha \sigma'$ (α-conversion).

4.1. Lemma (properties of $\preceq$). (1) $=_\alpha$ *is a congruence w.r.t.* $\preceq$, *i.e. if* $\sigma_1 =_\alpha \sigma_1'$ *and* $\sigma_2 =_\alpha \sigma_2'$, *then* $\sigma_1 \preceq \sigma_2$ *implies* $\sigma_1' \preceq \sigma_2'$;

(2) *If* $\sigma_1 \preceq \sigma_2$ *and* $\sigma_2 \preceq \sigma_3$, *then* $\sigma_1 \preceq \sigma_3$;

(3) $\sigma_1 =_\alpha \sigma_2$ *iff* $\sigma_1 \preceq \sigma_2$ *and* $\sigma_2 \preceq \sigma_1$;

(4) *If* $\sigma_1 \preceq \sigma_2$ *and* $\boldsymbol{\tau}$ *are open, then* $\sigma_1[\boldsymbol{\tau}/\boldsymbol{\alpha}] \preceq \sigma_2[\boldsymbol{\tau}/\boldsymbol{\alpha}]$.

Proof. Left to the reader. □

We consider two polymorphic typing disciplines, one of which is just the Damas–Milner system for *ML*, see [4], the other being a slightly modified version of this system. As in Section 3, each of these systems defines a programming language – we use the names *ML* and *EML* (the latter stands for "extended *ML*"). We use similar notation as for finitely typed expressions, i.e. "$E \vdash_{ML} M : \sigma$" and "$E \vdash_{EML} M : \sigma$" stand for "$M$ has the type σ in the environment E" (in the appropriate system). An obvious difference is that an environment E consists now of type assumptions of the form $(x : \sigma)$, where σ is an arbitrary type. If $E = \{(x_i : \sigma_i) \mid i \in I\}$, then $FV(E) = \bigcup \{FV(\sigma_i) \mid i \in I\}$, and by $E[\boldsymbol{\tau}/\boldsymbol{\alpha}]$ we denote the environment $E = \{(x_i : \sigma_i[\boldsymbol{\tau}/\boldsymbol{\alpha}]) \mid i \in I\}$ obtained by substituting $\boldsymbol{\tau}$ for all free $\boldsymbol{\alpha}$ in every type in E. The α-conversion on environments, denoted as $E =_\alpha E'$, is defined appropriately.

The type assertion system for *ML* consists of the following rules (here σ is an arbitrary type and τ, τ' range over open types):

(CONST) $E \vdash c : \tau$, where τ is the (finite) type of a constant c.

(VAR) $E \vdash x : \sigma$, if $(x : \sigma)$ is in E.

(APP) $$\frac{E \vdash M : \tau \to \tau',\ E \vdash N : \tau}{E \vdash (MN) : \tau'}.$$

(COND) $$\frac{E \vdash B : \mathbf{Bool},\ E \vdash M : \tau,\ E \vdash N : \tau}{E \vdash (\mathbf{if}\ B\ \mathbf{then}\ M\ \mathbf{else}\ N) : \tau}.$$

(ABS) $$\frac{E(x : \tau) \vdash M : \tau'}{E \vdash (\lambda x.\, M) : \tau \to \tau'}.$$

(FIX) $$\frac{E(x : \tau) \vdash M : \tau}{E \vdash (\mathbf{fix}\, x.\, M) : \tau}.$$

(LET) $$\frac{E \vdash N : \sigma,\ E(x : \sigma) \vdash M : \tau'}{E \vdash (\mathbf{let}\, x = N\ \mathbf{in}\ M) : \tau'}.$$

(GEN) $$\frac{E \vdash M : \sigma}{E \vdash M : \forall \alpha.\, \sigma},\ \text{where } \alpha \notin FV(E).$$

(INST) $$\frac{E \vdash M : \sigma}{E \vdash M : \sigma'},\ \text{where } \sigma \preceq \sigma'.$$

Note that here the only source of polymorphism in the above system is the LET rule.

The following are basic properties of this typing system. (Here we write for simplicity just "$\vdash$" instead of "$\vdash_{ML}$".)

4.2. Lemma. (1) *If $\sigma' =_\alpha \sigma$ and $E' =_\alpha E$, then $E \vdash M : \sigma$ implies $E' \vdash M : \sigma'$. That is, typability is invariant under α-conversion.*

(2) *For every term M and every environment E, if M is typable in E, then there exists a type σ such that*

- *$E \vdash M : \sigma$;*
- *$\sigma \preceq \sigma'$, for every σ' such that $E \vdash M : \sigma'$.*

The type σ is called the principal type for M in E.

(3) *If $E \vdash M : \sigma$ and $E \vdash T : E(F)$ for all "$F = T$" in a set of definitions D, then $\mathscr{A}$, $D \models M \twoheadrightarrow M'$ implies $E \vdash M' : \sigma$. That is, types are preserved by reductions.*

Proof. (1) is an obvious consequence of the rule INST and Lemma 4.1(3). The principal type property (2) has been proved by Damas and Milner ([4]). For the proof of (3), see Lemma A.4. □

As in the monomorphic case, a *program* in ML is a closed λ-term M such that

$$\vdash_{ML} M : \mathbf{0}^k \to \mathbf{0}$$

holds for some k. Again, M defines in every Σ-structure $\mathscr{A}$ a k-ary partial function

$$M^{\mathscr{A}}: \mathscr{A}^k \to \mathscr{A}$$

defined by

$$M^{\mathscr{A}}(a_1, \ldots, a_k) = a \text{ iff } \mathscr{A} \models Ma_1 \ldots a_k \twoheadrightarrow a.$$

A semiprogram (D, M) is *typable* in ML iff there exists an environment E, such that
- $E \vdash_{ML} M : \sigma$, for some σ;
- $E \vdash_{ML} T : E(F)$, for all "$F = T$" in D;
- all types in E are open.

Then (D, M) is said to have the type σ in E.

A *recursive procedure* in ML is a semiprogram (D, M) of type $\mathbf{0}^k \to \mathbf{0}$, such that M is an applicative term and the definitions of D are simple.

4.3. Lemma. *For each ML-program there exists an equivalent recursive procedure in ML.*

Proof. See Lemma A.5. □

The above result leads directly to a characterization of the expressive power of ML. Indeed, a recursive procedure in ML is just a system of recursive definitions, involving no truly polymorphic functions (of universal types). The reduction of an arbitrary ML-program to such an essentially monomorphic construction is possible because the only source of polymorphism in ML is the constructor **let**. By repeatedly "unwinding" **let** (see Transformation I), we can entirely get rid of all occurrences of this construct. (The situation with EML is different: see Example 5.3.)

4.4. Lemma. *Let E be an environment that assigns only monomorphic types, and let τ be a monomorphic type. For an arbitrary applicative term M, if $E \vdash_{ML} M : \tau$, then $E \vdash_{FL} M : \tau$.*

Proof. The derivation $E \vdash_{ML} M : \tau$ may in general use type assertions that involve types with variables. However, in Lemma A.2 we show that the derivation may be assumed not to use rules GEN and INST (it does not use LET since no **let** occurs in M). Thus, by an easy induction we conclude that all types used in such a derivation are in fact monomorphic, which means that $E \vdash_{FL} M : \tau$. □

4.5. Theorem. $ML \equiv FL$.

Proof. We transform an arbitrary ML-program into a recursive procedure (D, M) in ML. Let E be the appropriate environment and let E^0 be obtained from E by replacing all variables free in E by the constant $\mathbf{0}$. Then, by Lemma 4.2, we have

$E^0 \vdash_{ML} M : \mathbf{0}^k \to \mathbf{0}$ and $E^0 \vdash_{ML} T : E^0(F)$, for all "$F = T$" in D. The hypothesis follows now from Lemma 4.4. □

Thus, the problem of the expressive power of ML is settled, and we can concentrate on the polymorphic programs of EML.

5. Universally polymorphic recursive procedures

The type assertion system for EML differs from that for ML in that the rule (FIX) is replaced by the following rule:

$$(\mathrm{FIX}^+) \qquad \frac{E(x:\sigma) \vdash M : \sigma}{E \vdash (\mathbf{fix}\, x.\, M) : \sigma}.$$

That is, in EML the fixpoint rule may be applied to polymorphic types. Throughout this section the symbol "$\vdash$" stands for "$\vdash_{EML}$". We start with a technical lemma, an analogue of Lemma 4.2.

5.1. Lemma. (1) *If $\sigma' =_\alpha \sigma$ and $E' =_\alpha E$, then $E \vdash M : \sigma$ implies $E' \vdash M : \sigma'$.*

(2) *For every term M and every environment E, if M is typable in E, then there exists a type σ such that*
- *$E \vdash M : \sigma$;*
- *$\sigma \preceq \sigma'$ for every σ' such that $E \vdash M : \sigma'$.*

The type σ is called the prrincipal type for M in E.

(3) *If $E \vdash M : \sigma$ and $E \vdash T : E(F)$ for all "$F = T$" in a set of definitions D, then $\mathscr{A}, D \models M \twoheadrightarrow M'$ implies $E \vdash M' : \sigma$.*

Proof. (1) is easy, (2) is due to [30], and for the proof of (3), see Lemma A.4. □

A semiprogram (D, M) is *typable* in EML iff there exists an environment E such that
- $E \vdash_L M : \sigma$, for some σ;
- $E \vdash_L T : E(F)$, for all "$F = T$" in D;
- all types in E are closed.

Then (D, M) is said to have the type σ in E.

The notions of a *program* and *recursive procedure* are defined in a similar way as for ML. These notions are equivalent with respect to computational power.

5.2. Lemma. *For each recursive procedure in EML there exists an equivalent EML-program, and conversely.*

Proof. See Lemmas A.5 and A.6. □

To illustrate the difference between ML and EML, consider the following example.

5.3. Example (Urzyczyn [38]). Let Σ be as in Example 3.3 and let $P=(D, \mathscr{F})$, where D consists of the following function definitions:

$$\mathscr{F} = \lambda xy.\mathscr{G}xgy,$$

$$\mathscr{G} = \lambda xF Y.\textbf{ if } rx \textbf{ then } FY \textbf{ else } \mathscr{G}(fx)\mathscr{H}FY,$$

$$\mathscr{H} = \lambda FX.F(FX).$$

We have $E \vdash_{EML} P : \mathbf{0}^2 \to \mathbf{0}$, for the environment E defined below:

$$\mathscr{F} : \mathbf{0}^2 \to \mathbf{0},$$

$$\mathscr{G} : \forall\alpha.\, \mathbf{0} \to (\alpha \to \alpha) \to \alpha \to \alpha,$$

$$\mathscr{H} : \forall\alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha.$$

If $(\mathscr{A}, (a, b))$ is as in Example 3.3, then $\mathscr{A}$, $D \models \mathscr{F}ab \twoheadrightarrow g^{e(n)}b$, where $e(0)=1$ and $e(m+1)=2^{e(m)}$, for all m. Note that $e(n)$ is a nonelementary function. Note also that this recursive procedure is neither typable in FL nor in ML, and that during the computation one has to apply $\mathscr{G}$ to various finitely typed arguments with more and more complex types.

We show here that EML is a universal programming language, i.e. it is equivalent to the class of all flowcharts with counters and ordinary pushdown stores. Clearly, computing with counters in a Σ-structure $\mathscr{A}$ is equivalent to computing in a 2-sorted structure consisting of $\mathscr{A}$ and the standard model of arithmetic. Thus, an easy modification of the classical result of [1, 3] (or of our Theorem 3.5, for $k=1$) shows that each program with a pushdown store and counters may be seen as a recursive procedure of level 1 over an appropriate many-sorted signature.

Let us state this observation more precisely. First note that our untyped language, as well as the typed ones may equally well be defined with respect to a many-sorted signature rather than an ordinary signature, and all properties considered in the previous sections remain unchanged. Of course, the notion of a type must be redefined by allowing a separate type constant for each sort. Now if Σ is a first-order signature, we may consider a 2-sorted signature $\Sigma_{\mathbf{Int}}$ obtained from Σ by adding the operation symbols 0 and *succ* (0-ary and unary) over an additional sort (for integers). According to our convention, $\Sigma_{\mathbf{Int}}$ must also contain an equality symbol for the new sort, denoted as *eq*. Let **Int** be a new base type. The types associated with the new constants are, of course,

$$0 : \mathbf{Int},$$

$$eq : \mathbf{Int}^2 \to \mathbf{Bool},$$

$$succ : \mathbf{Int} \to \mathbf{Int}.$$

The syntax and semantics of the (generalized) λ-terms over $\Sigma_{\mathbf{Int}}$ is defined appropriately (with respect to an arbitrary 2-sorted Σ_{Int}-structure). If $\mathscr{A}$ is a Σ-structure, then $\mathscr{A}$ can be extended to a model of $\Sigma_{\mathbf{Int}}$ by specifying the interpretation for the new sort. The most natural extension is to choose the set ω of nonnegative integers, interpreting 0 and *succ* as zero and the ordinary successor. This is the default extension, denoted as $\mathscr{A}_\omega$. For technical reasons, we need to consider also another possibility: that the type **Int** corresponds to an initial segment of integers $m=\{0,\ldots,m-1\}$ with the successor operation modified so that $succ(m-1)=m-1$. This $\Sigma_{\mathbf{Int}}$-structure is denoted as $\mathscr{A}_m$. We use the following notation:

$$\mathscr{A}\models M\to N, \text{ for } \mathscr{A}_\omega\models M\to N;$$

$$\mathscr{A}\models_m M\to N, \text{ for } \mathscr{A}_m\models M\to N.$$

A *recursive scheme with arithmetic* is a simple semiprogram $P=(D, M)$, over the extended language, which is typable in FL_1, so that the type assigned to M is of the form $\mathbf{0}^k\to\mathbf{0}$. If $\mathscr{A}$ is a Σ-structure then we define $P^{\mathscr{A}}:\mathscr{A}^k\to\mathscr{A}$ by

$$P^{\mathscr{A}}(\boldsymbol{a})=b \text{ iff } \mathscr{A}, D\models M\boldsymbol{a}\twoheadrightarrow b$$

and, thus, P may be considered to be a program over Σ. By the above remarks it is easy to see that each flowchart with counters and a pushdown store is equivalent to a recursive scheme with arithmetic and, thus, our goal is to show that the latter can be simulated with an EML-program.

This task will be divided into several steps. The first one is to show that, for an arbitrary $\mathscr{A}$, derivations in $\mathscr{A}_{2^m}$ can be simulated by derivations in $\mathscr{A}_m$. We start with a notational convention: for arbitrary open types τ and σ, let $\vec{\tau}\to\sigma$ stand for the type

$$(\tau\to\tau)\to(\tau^2\to\mathbf{Bool})\to\tau\to\sigma.$$

Assume that D is a fixed set of simple function definitions not involving occurrences of *succ*, *eq* and 0, and let $\mathscr{Z}$, $\mathscr{S}$, $\mathscr{S}^*$, $\mathscr{S}^{**}$, $\mathscr{R}$, $\mathscr{R}^*$ be new function variables. Denote by E_{exp} an environment consisting of the following type assumptions:

$$\mathscr{Z}:\vec{\alpha}\to\alpha\to\mathbf{Bool},$$

$$\mathscr{S}:\vec{\alpha}\to(\alpha\to\mathbf{Bool})\to\alpha\to\mathbf{Bool},$$

$$\mathscr{S}^{**}, \mathscr{S}^*:\vec{\alpha}\to(\alpha\to\mathbf{Bool})\to\alpha\to\alpha\to\mathbf{Bool},$$

$$\mathscr{R}:\vec{\alpha}\to(\alpha\to\mathbf{Bool})\to(\alpha\to\mathbf{Bool})\to\mathbf{Bool},$$

$$\mathscr{R}^*:\vec{\alpha}\to\alpha\to(\alpha\to\mathbf{Bool})\to(\alpha\to\mathbf{Bool})\to\mathbf{Bool}.$$

Let D_{exp} be the following set of function definitions:

$$\mathscr{Z} = \lambda sezx.false,$$

$$\mathscr{S} = \lambda sezXx.\mathscr{S}^* sezXzx,$$

$$\mathscr{S}^* = \lambda sezXyx.\textbf{if } ey(sy) \textbf{ then } true$$
$$\textbf{else if } Xy \textbf{ then } \mathscr{S}^* sez(\mathscr{S}^{**}Xy)(sy)x$$
$$\textbf{else } \mathscr{S}^{**}Xyx,$$

$$\mathscr{S}^{**} = \lambda sezXyx.\textbf{if } eyx \textbf{ then}$$
$$\textbf{if } Xy \textbf{ then } false \textbf{ else } true$$
$$\textbf{else } Xx,$$

$$\mathscr{R} = \lambda sezXY.\mathscr{R}^* sezzXY,$$

$$\mathscr{R}^* = \lambda sezxXY.\textbf{if } ex(sx) \textbf{ then}$$
$$\textbf{if } eq^{\textbf{Bool}}(Xx)(Yx) \textbf{ then } true \textbf{ else } false$$
$$\textbf{else if } eq^{\textbf{Bool}}(Xx)(Yx) \textbf{ then } \mathscr{R}^* sez(sx)XY$$
$$\textbf{else } false.$$

The definitions of D_{exp} are typable in FL in the environment E_{exp}. One can also easily see that the same holds for $E_{exp}[\tau/\alpha]$, for an arbitrary open type τ. Also, D_{exp} is typable in EML with respect to the environment $E^{\forall}_{exp}$, obtained by universally quantifying all types in E_{exp}.

Let $0^\bullet$ be the term $\mathscr{Z}\ succ\ eq\ 0$, and for $n>0$, let $n^\bullet \equiv \mathscr{S}\ succ\ eq\ 0(n-1)^\bullet$. Now, let T be an arbitrary generalized applicative term over the signature $\Sigma_{\textbf{Int}}$. By $T^\bullet$ we denote a term obtained from T by replacing each occurrence of

$succ$	by	$\mathscr{S}\ succ\ eq\ 0$
eq	by	$\mathscr{R}\ succ\ eq\ 0$
0	by	$0^\bullet$

and replacing each algebraic constant $n>0$ by $n^\bullet$.

Let us fix a number $m>1$. For $n<2^m$ and $i<m$, let $\beta(i, n)\in\{0, 1\}$ be such that

$$n = \beta(m-1, n)\cdot 2^{m-1} + \cdots + \beta(1, n)\cdot 2 + \beta(0, n),$$

and let $b(i, n)$ be *true* if $\beta(i, n)=1$ and *false* otherwise.

5.4. Lemma. (1) *For all* $i<m$, $\mathscr{A}$, $D\cup D_{exp}\models_m \mathscr{Z}\ succ\ eq\ 0\ i \twoheadrightarrow 0$.

(2) *Assume that* $n<2^m$ *and* $\mathscr{A}$, $D\models_m M_i \twoheadrightarrow b(i, n)$, *for all* $i<M$. *Then*

$$\mathscr{A},\ D\cup D_{exp}\models_m \mathscr{S}\ succ\ eq\ 0\ M\ i \twoheadrightarrow b(i, \min(n+1, 2^m-1)).$$

In particular, it follows that $\mathscr{A}$, $D\cup D_{exp}\models_m n^\bullet\ i \twoheadrightarrow b(i, n)$, *for all* $n<2^m$.

(3) *Let* M_1 *and* M_2 *be such that, for all* $i<m$, *and some* $n_1, n_2<2^m$

$$\mathscr{A}, D\models_m M_1\, i\twoheadrightarrow b(i, n_1),$$

$$\mathscr{A}, D\models_m M_2\, i\twoheadrightarrow b(i, n_2).$$

Then $\mathscr{A}, D\cup D_{exp}\models_m \mathscr{R}\, succ\, eq\, 0\, M_1\, M_2\twoheadrightarrow b$, *where* $b\equiv true$ *if* $n_1=n_2$ *and* $b\equiv false$ *otherwise.*

Proof. In order to make it easier to understand the definitions in D_{exp} we give an informal explanation. Below we use the same notation for formal parameters as in the above definitions. The reader should view functions of type **Int**→**Bool** (denoted with capital letters) as binary arrays indexed by numbers in $\{0,\ldots, m-1\}$ (denoted with lower case characters) or just as binary expansions of numbers in $\{0,\ldots, 2^m-1\}$. (Here α is instantiated to **Int**.) Then

- $\mathscr{Z}$ represents the array of 0's;
- $\mathscr{S}$ adds one to the array X;
- $\mathscr{R}$ tests equality of arrays X and Y;
- $\mathscr{R}^*$ tests equality between X and Y from the xth place on;
- $\mathscr{S}^*$ adds X and the array consisting of all 0's, except on the yth place it has 1;
- $\mathscr{S}^{**}$ reverses the yth bit in X.

The details of the proof are left to the reader. □

The next lemma shows that one can exponentiate the size of available integer values if necessary.

5.5. Lemma. *Let* D *be a set of function definitions over* Σ, *i.e. with no occurrences of succ, eq and* 0. *Assume* D *to be FL-typable with respect to an environment* E, *and let* T *be a* D*-closed generalized applicative term over* $\Sigma_{\mathbf{Int}}$, *of a base type in* E. *Then*

(1) *for each* $a\in\mathscr{A}\cup\{true, false\}$

$$\mathscr{A}, D\models_{2^m} T\twoheadrightarrow a \text{ iff } \mathscr{A}, D\cup D_{exp}\models_m T^\bullet\twoheadrightarrow a,$$

(2) *for each* $n\in\{0,\ldots, m-1\}$

$$\mathscr{A}, D\models_{2^m} T\twoheadrightarrow n \text{ iff } \mathscr{A}, D\cup D_{exp}\models_m T^\bullet i\twoheadrightarrow b(i, n), \text{ for all } i<m.$$

Proof. We show (1) and (2) simultaneously. For the "only if" part, we proceed by induction with respect to the length of the derivation $\mathscr{A}, D\models_{2^m} T\twoheadrightarrow_{\mathrm{L}} a$ in the case of (1), or $\mathscr{A}, D\models_{2^m} T\twoheadrightarrow_{\mathrm{L}} n$ in the case of (2). Consider first the case when the first step

$$\mathscr{A}, D\models_{2^m} T\to_{\mathrm{L}} T_1$$

is obtained by reducing a redex not of the form "$succ\, n$" or "$eq\, n_1\, n_2$". Then, by inspecting all possibilities, one finds easily that

$$\mathscr{A}, D\cup D_{exp}\models_m T^\bullet\to T_1^\bullet,$$

and it suffices to apply the induction hypothesis for T_1. Assume then that the leftmost redex located in T is "$succ\, n$" or "$eq\, n_1\, n_2$". Then T must have the form $M_1 \cdots M_k$, where M_1 is not an application and $k \geqslant 1$. (Note that M_1 cannot be an abstraction – if $k > 1$, then the leftmost redex is a β-redex; if $k = 1$, then T cannot be of a base type.)

Case 1: M_1 has the form **if** P **then** Q **else** R. Now, P is not a boolean constant (otherwise "**if**" would be leftmost) and, thus, the leftmost redex is a part of P. But P must reduce to a constant and we have either $\mathscr{A}, D \models_{2^m} P \twoheadrightarrow_{\mathrm{L}} true$ or $\mathscr{A}, D \models_{2^m} P \twoheadrightarrow_{\mathrm{L}} false$. Using the induction hypothesis for P, we obtain $\mathscr{A}, D \cup D_{exp} \models_m T^\bullet \twoheadrightarrow Q^\bullet M_2^\bullet \cdots M_k^\bullet$ in case of *true*, and similarly for *false*. Then we use the induction hypothesis once more for $QM_2 \cdots M_k$.

If Case 1 does not hold, then M_1 must be an identifier. It is not a variable since T is D-closed and defined variables are redexes.

Case 2: M_1 is a symbol from Σ, say f. Then f must be $(k-1)$-ary, and $T \equiv f a_1 \cdots a_{i-1} M_i \cdots M_k$, with the leftmost redex located in M_i. As before we apply the induction hypothesis first for M_i, and then for the term obtained from T by reducing M_i to a constant.

Case 3: M_1 is *succ*, in which case $k = 2$, and we have $T \equiv succ\, T_1$; in particular, T reduces to an integer. It must hold that $\mathscr{A}, D \models_{2^m} T_1 \twoheadrightarrow_{\mathrm{L}} n$, for some $n < 2^m$. By the induction hypothesis, $\mathscr{A}, D \cup D_{exp} \models_m T_1^\bullet\, i \twoheadrightarrow b(i, n)$, for an arbitrary $i < m$. It remains to use Lemma 5.4(2).

Case 4: M_1 is *eq*. Then $k = 3$ and $T \equiv eq\, T_1\, T_2$. We have $\mathscr{A}, D \models_{2^m} T_1 \twoheadrightarrow_{\mathrm{L}} n_1$ and $\mathscr{A}, D \models_{2^m} T_2 \twoheadrightarrow_{\mathrm{L}} n_2$, with T reducing to *true* or *false*, depending on whether $n_1 = n_2$ or not. The hypothesis follows from Lemma 5.4(3) and the induction hypotheses for T_1 and T_2.

Case 5: M_1 is 0 and $k = 1$. The hypothesis follows from Lemma 5.4(1).

Now turn to the proof of the "if" part. Since T is of a base type, a normal form of T (if it exists) must be a constant. After the "only if" part has been shown, it suffices to prove that if $T^\bullet$ or $T^\bullet 0$ reduces to a constant with respect to $\mathscr{A}_m$ and $D \cup D_{exp}$, then also T reduces to a constant with respect to $\mathscr{A}_{2^m}$ and D. We proceed by induction on the derivation $\mathscr{A}, D \cup D_{exp} \models_m T^\bullet \twoheadrightarrow_{\mathrm{L}} a$ in the case of (1) (or $\mathscr{A}, D \cup D_{exp} \models_m T^\bullet \mathbf{0} \twoheadrightarrow_{\mathrm{L}} b \in \{true, false\}$ in the case of (2)). The term T can always be presented as $T = M_1 \cdots M_k$, where M_1 is not an application.

Case 0: M_1 is an abstraction. Then $k \neq 1$, otherwise T cannot be of a base type. But then the leftmost redex is the β-redex $M_1 M_2$, and for some T_1, we have $\mathscr{A}, D \models_{2^m} T \twoheadrightarrow_{\mathrm{L}} T_1$ and $\mathscr{A}, D \cup D_{exp} \models_m T^\bullet \twoheadrightarrow_{\mathrm{L}} T_1^\bullet$. Thus, it suffices to use the induction hypothesis for T_1.

Case 1: $M_1 \equiv$ **if** P **then** Q **else** R. Since $P^\bullet$ must reduce to a boolean constant, say *true*, the term P must also reduce to *true* by the induction hypothesis. Then T reduces to $QM_2 \cdots M_k$, while $T^\bullet$ reduces to $Q^\bullet M_2^\bullet \cdots M_k^\bullet$. We use the induction hypothesis for $QM_2 \cdots M_k$ (or $RM_2 \cdots M_k$, if P reduces to *false*).

Case 2: M_1 is a symbol from Σ or a defined variable. The proof for this case is an easy application of the induction hypothesis.

Case 3: M_1 is *succ*, i.e. $T \equiv succ\, T_1$, with $T^\bullet 0$ reducing to a boolean constant. An easy inspection of the definition of $\mathscr{S}$ shows that a successful evaluation of $\mathscr{S}\, succ\, eq\, 0\, T_1^\bullet 0$ requires a successful evaluation of $T_1^\bullet 0$. Thus, T_1 must reduce to an integer by the induction hypothesis, and the same holds for T.

Case 4: M_1 is *eq*, i.e. $T \equiv eq\, T_1\, T_2$. Since $T^\bullet \equiv \mathscr{R}\, succ\, eq\, 0\, T_1^\bullet T_2^\bullet$ reduces to a boolean value, it must be the case that both $T_1^\bullet 0$ and $T_2^\bullet 0$ have normal forms (by the definition of $\mathscr{R}$). It follows that T_1 and T_2 have normal forms and, thus, T successfully reduces to a boolean. □

We need another technical definition. Let $P=(D, M)$ be a simple semiprogram over $\Sigma_{\mathbf{Int}}$, and let s, e, z be variables not occurring in P. We construct a new semiprogram $\bar{P}=(\bar{D}, \bar{M})$ as follows. For each term N, let N' be obtained from N by
- replacing each occurrence of *succ*, *eq*, 0 in N by s, e, z, respectively;
- replacing each occurrence of a variable $\mathscr{F}$ defined in D, by $\mathscr{F} sez$.

We set $\bar{M} \equiv \lambda sez.\, M'$ and let $\bar{D}$ consist of definitions of the form $\mathscr{F} = \lambda sez.\, N'$, for all "$\mathscr{F} = N$" in D.

5.6. Lemma. Let $P=(D, M)$ *be as above and let* τ *be an open type.*

(1) *If* $E \vdash P : \sigma$, *with* σ *open, then* $\bar{E} \vdash \bar{P} : \dot{\tau} \to \sigma$, *where* $\bar{E}(\mathscr{F}) = \dot{\tau} \to E(\mathscr{F})$, *for all defined variables* $\mathscr{F}$.

(2) *If* σ *above has the form* $\mathbf{0}^k \to \mathbf{0}$, *then for an arbitrary* Σ*-interpretation* $(\mathscr{A}, \boldsymbol{a})$ *and any* $b \in \mathscr{A}$ *it holds that*

$$\mathscr{A}, D \models_{(m)} M\boldsymbol{a} \twoheadrightarrow b \text{ iff } \mathscr{A}, \bar{D} \models_{(m)} \bar{M}\, succ\, eq\, 0\, \boldsymbol{a} \twoheadrightarrow b.$$

Proof. Left to the reader – note that (2) corresponds to the case $\tau = \mathbf{Int}$ in (1). □

The next definition is crucial for our consideration. Let $m \in \omega$, let τ be a finite type over $\{\mathbf{0}, \mathbf{Bool}\}$, let D_ω be a finite set of function definitions, and let S, R, Z be terms. The quadruple (D_ω, S, R, Z) is said to *define an arithmetic of type* τ *and size* m in a k-interpretation $(\mathscr{A}, \boldsymbol{a})$ iff, for some environment E_ω,

(1) $E_\omega \vdash (D_\omega, S) : \tau \to \tau$;

(2) $E_\omega \vdash (D_\omega, R) : \tau \to \tau \to \mathbf{Bool}$;

(3) $E_\omega \vdash (D_\omega, Z) : \tau$,

and for every set D of simple function definitions, not involving *succ*, *eq* and 0, every term M over Σ and every $b \in \mathscr{A}$:

(4) $\mathscr{A}, D \models_m M\, succ\, eq\, 0 \boldsymbol{a} \twoheadrightarrow b$ iff $\mathscr{A}, D \cup D_\omega \models M S R Z \boldsymbol{a} \twoheadrightarrow b$.

That is, S, R, Z play the role of *succ*, *eq* and 0 unless the integer values used are greater than or equal m. Note that (4) is equivalent to the following statement. For every M over Σ with $FV(M) = \{s, e, z\}$,

(4′) $\mathscr{A}, D \models_m M[succ/s, eq/e, 0/z]\boldsymbol{a} \twoheadrightarrow b$ iff $\mathscr{A}, D \cup D_\omega \models M[S/s, R/e, Z/z]\boldsymbol{a} \twoheadrightarrow b$.

The following is a simple consequence of the above definition and Lemma 5.6.

5.7. Lemma. *Let (D_ω, S, R, Z) define arithmetic of type τ and size m in $(\mathscr{A}, \boldsymbol{a})$. Then for every simple semiprogram $P=(D, M)$ over $\Sigma_{\mathbf{Int}}$*

$$\mathscr{A}, D \models_m M\boldsymbol{a} \twoheadrightarrow b \text{ iff } \mathscr{A}, \bar{D} \cup D_\omega \models \bar{M} S R Z \boldsymbol{a} \twoheadrightarrow b.$$

Using Lemma 3.4, we can now define arithmetic of type **0**.

5.8. Lemma. *Let $P_{Next}=(D_{Next}, Next)$ be the program of arity k provided by Lemma 3.4, and let $(\mathscr{A}, \boldsymbol{a})$, with $\boldsymbol{a}=(a_1, \ldots, a_k)$, be an arbitrary k-interpretation. Then $(D_{Next}, Next\boldsymbol{a}, eq^{\mathbf{0}}, a_1)$ defines arithmetic of type* **0** *and size $|\mathscr{A}|$ in $(\mathscr{A}, \boldsymbol{a})$.*

Proof. Left to the reader. □

Thus, for a finite interpretation of size n we can now simulate counters of size n, and we may increase this bound using the technique of Lemma 5.5.

5.9. Lemma. *Let $(\mathscr{A}, \boldsymbol{a})$ be an arbitrary k-interpretation. If D_ω and D_{exp} have disjoint defined variables and if (D_ω, S, R, Z) defines arithmetic of size m and type τ in $(\mathscr{A}, \boldsymbol{a})$, then $(D_\omega \cup D_{exp}, \mathscr{S}SRZ, \mathscr{R}SRZ, \mathscr{Z}SRZ)$ defines arithmetic in $(\mathscr{A}, \boldsymbol{a})$ of size 2^m and type $\tau \to$* **Bool**.

Proof. The reader can easily check if E_ω is an environment satisfying the conditions (1)–(3) above; then $\mathscr{S}SRZ$, $\mathscr{R}SRZ$, $\mathscr{Z}SRZ$ are assigned, respectively, the types $(\tau \to \mathbf{Bool}) \to (\tau \to \mathbf{Bool})$, $(\tau \to \mathbf{Bool})^2 \to \mathbf{Bool}$ and $\tau \to \mathbf{Bool}$, in the environment $E_{exp}[\tau/\alpha] \cup E_\omega$. It remains to show that

$$\mathscr{A}, D \models_{2^m} M\, succ\, eq\, 0\, \boldsymbol{a} \twoheadrightarrow b$$

$$\text{iff } \mathscr{A}, \bar{D} \cup D_{exp} \cup D_\omega \models \bar{M}(\mathscr{S}SRZ)(\mathscr{R}SRZ)(\mathscr{Z}SRZ)\boldsymbol{a} \twoheadrightarrow b,$$

for every semiprogram $P=(D, M)$ and $b \in \mathscr{A}$, provided D, M do not involve *succ*, *eq* or 0. Since $(D_\omega \cup D_{exp}, \mathscr{S}SRZ, \mathscr{R}SRZ, \mathscr{Z}SRZ)$ defines arithmetic, the statement at the right-hand side is equivalent to

$$\mathscr{A}, \bar{D} \cup D_{exp} \cup D_\omega \models_m \bar{M}(\mathscr{S}\, succ\, eq\, 0)(\mathscr{R}\, succ\, eq\, 0)(\mathscr{Z}\, succ\, eq\, 0)\boldsymbol{a} \twoheadrightarrow b.$$

This in turn is equivalent to $\mathscr{A}, D \models_{2^m} M\, succ\, eq\, 0\, \boldsymbol{a} \twoheadrightarrow b$, by Lemma 5.5. □

Let $\tau_0 = \alpha$ and let $\tau_{i+1} = \tau_i \to$ **Bool**, for $i \geqslant 0$. It is a straightforward consequence of Lemma 5.9 that, for each $l \geqslant 1$, there are terms $\mathscr{S}_l$, $\mathscr{R}_l$, $\mathscr{Z}_l$, such that if (D_ω, S, R, Z) defines arithmetic of size m and type τ in $(\mathscr{A}, \boldsymbol{a})$, then $(D_\omega \cup D_{exp}, \mathscr{S}_l SRZ, \mathscr{R}_l SRZ, \mathscr{Z}_l SRZ)$ defines arithmetic in $(\mathscr{A}, \boldsymbol{a})$ of size $exp_l(m)$ and type $\tau_l[\tau/\alpha]$.

Putting together Lemmas 5.8 and 5.9 enables us to simulate counters provided their values are bounded by an elementary recursive function in the size of an interpretation. Let us make precise the notion of the size of counters necessary to perform

a successful computation. Let D be a set of definitions and let M be a term. We say that M *uses* a value $m \in \omega$ with respect to $\mathscr{A}$ and D iff for some N such that $\mathscr{A}, D \models M \twoheadrightarrow_L N$, the algebraic constant m occurs in N. Let (D, M) be a semiprogram of type $\mathbf{0}^k \to \mathbf{Int}^l \to \mathbf{a}$, for some base type $\mathbf{a}$. If $\mathscr{A}, D \models M\,\boldsymbol{a}\,\boldsymbol{n} \twoheadrightarrow_L a$, where a is a constant of type $\mathbf{a}$, and $M\,\boldsymbol{a}\,\boldsymbol{n}$ uses only values less than m, with respect to $\mathscr{A}$ and D, then we say that m is a *bound* for (D, M) in $(\mathscr{A}, \boldsymbol{a}, \boldsymbol{n})$. In this case we can easily see that $\mathscr{A}, D \models_m M\,\boldsymbol{a}\,\boldsymbol{n} \twoheadrightarrow_L a$ as well.

We turn to our main task of showing that an arbitrary recursive scheme with arithmetic $P = (D, M)$ may be simulated by an EML-program. Let us fix P as above and assume P to be of type $\mathbf{0}^k \to \mathbf{0}$. We want to determine bounds for P in finite interpretations.

In Section 1 we considered codes of finite interpretations. It is more convenient now to assume that a code of an interpretation is a number $\kappa(\mathscr{I}) \in \omega$ rather than a sequence $c(\mathscr{I}) \in \{0, 1\}^*$. The number $\kappa(\mathscr{I}) \in \omega$ can be defined using any standard method of coding binary words. Note that, since $c(\mathscr{I})$ is of length at most polynomial in the size of $\mathscr{I}$, the value $\kappa(\mathscr{I})$ will be at most exponential, in particular, elementary, in the size of $\mathscr{I}$. A bound for P in $\mathscr{I}$ may now be seen as a function on $\kappa(\mathscr{I})$. Unfortunately, such a function will not in general be recursive. The best we can do is to observe that there exists a partial recursive function $\varphi : \omega \to \omega$ such that $\varphi(\kappa(\mathscr{A}, \boldsymbol{a}))$ is defined iff $\mathscr{A}, D \models M\,\boldsymbol{a} \twoheadrightarrow a$, for some $a \in \mathscr{A}$, and then $\varphi(\kappa(\mathscr{A}, \boldsymbol{a}))$ is a bound for P in $(\mathscr{A}, \boldsymbol{a})$. However, from recursion theory we know that there is an elementary recursive function $f : \omega^2 \to \omega$ such that, for all $n \in \mathrm{Dom}(\varphi)$,

$$(*) \qquad \varphi(n) \leqslant \mu y.(f(n, y) = 0).$$

5.10. Lemma. *Let f be an elementary recursive function that satisfies (*). There exists an $l \in \omega$ and a simple semiprogram $P_f = (D_f, \mathscr{F})$ of type $\forall\alpha.\vec{\alpha} \to \mathbf{0}^k \to \alpha \to \alpha \to \mathbf{Bool}$ with $\mathscr{F}$ a function variable, and such that for all finite k-interpretations $(\mathscr{A}, \boldsymbol{a})$, and all $p \in \omega$*

(1) $\mathscr{A}, D_f \models \mathscr{F}\ succ\ eq\ 0\ \boldsymbol{a}\ p \twoheadrightarrow true$ *iff* $f(\kappa(\mathscr{A}, \boldsymbol{a}), p) = 0$;

(2) $\mathscr{A}, D_f \models \mathscr{F}\ succ\ eq\ 0\ \boldsymbol{a}\ p \twoheadrightarrow false$ *iff* $f(\kappa(\mathscr{A}, \boldsymbol{a}), p) \neq 0$;

(3) $exp_l(|\mathscr{A}|, p)$ *is a bound for* P_f *in* $(\mathscr{A}, \boldsymbol{a}, p)$.

Proof. First note that computing $\kappa(\mathscr{A}, \boldsymbol{a})$ is fairly easy with the help of the semiprogram $(D_{Next}, Next)$ of Lemma 3.4. It is left to the reader to check that an appropriate semiprogram may be constructed, and to ensure that, for some r, $exp_r(|\mathscr{A}|)$ is a bound for this semiprogram in $(\mathscr{A}, \boldsymbol{a})$. The next step will be to provide function definitions so that computing $f(q, p)$ becomes possible for arbitrary q, p, and within an elementary bound in $\max(q, p)$. This task (again, details are left to the reader) may be achieved by an induction on the construction of f (from projections, addition and subtraction, with help of substitutions, indexed sums and products). □

Before we turn to the main construction, we need one more technical definition. Let $D_{\mathscr{P}}$ be the following set of function definitions (here, l is a number provided by

Lemma 5.10, and $\mathscr{S}_l$, $\mathscr{R}_l$, $\mathscr{Z}_l$ are the terms defined after Lemma 5.9):

$$\mathscr{L} = \lambda sezy.\ \textbf{if}\ ezy\ \textbf{then}\ \mathscr{Z}_l sez$$
$$\textbf{else}\ \mathscr{S}_l sez(\mathscr{L} sez(\mathscr{P} sezy)),$$
$$\mathscr{P} = \lambda sezx.\ \mathscr{P}^* sezzzx,$$
$$\mathscr{P}^* = \lambda sezyvx.\ \textbf{if}\ evx\ \textbf{then}\ y$$
$$\textbf{else if}\ evz\ \textbf{then}\ \mathscr{P}^* sezy(sz)x$$
$$\textbf{else}\ \mathscr{P}^* sez(sy)(sv)x.$$

One can easily see that $D_{\mathscr{L}}$: is EML-typable in the following environment.

$$\mathscr{L} : \forall\alpha.\ \vec{\alpha}\to\alpha\to\alpha\to\tau_l,$$
$$\mathscr{P} : \forall\alpha.\ \vec{\alpha}\to\alpha\to\alpha,$$
$$\mathscr{P}^* : \forall\alpha.\ \vec{\alpha}\to\alpha\to\alpha\to\alpha\to\alpha.$$

5.11. Lemma. *Let (D_ω, S, R, Z) define arithmetic of size m in $(\mathscr{A}, \boldsymbol{a})$. Then, for every $n<m$,*

$$\mathscr{A}, D_\omega \cup D_{\mathscr{L}} \models_m \mathscr{L} SRZ(S^n Z) \twoheadrightarrow (\mathscr{S}_l SRZ)^n (\mathscr{Z}_l SRZ).$$

Proof. First note that $\mathscr{P}$ represents predecessor in the following sense. For every $n<m-1$,

$$\mathscr{A}, D_\omega \cup D_{\mathscr{L}} \models \mathscr{P} SRZ(S^{n+1}Z) \twoheadrightarrow (S^n Z) \quad \text{and} \quad \mathscr{A}, D_\omega \cup D_{\mathscr{L}} \models \mathscr{P} SRZZ \twoheadrightarrow Z.$$

For this it suffices to note that for arbitrary n_1, n_2, the term $R(S^{n_1}Z)(S^{n_2}Z)$ reduces to the same boolean value as $eq\, n_1\, n_2$ (in $\mathscr{A}_m$), because (D_ω, S, R, Z) defines arithmetic of size m. Thus, $\mathscr{L} SRZ$ applied to $S^n Z$ will generate n copies of $\mathscr{S}_l SRZ$. □

We are now ready to state our main lemma.

5.12. Lemma. *There exists an EML recursive procedure P_{fin} which is equivalent to P in all finite k-interpretations of size at least 2.*

Proof. Recall that $P=(D, M)$ and choose D_{Next}, D_{exp}, $D_{\mathscr{L}}$, D_f so that their defined variables are disjoint and do not occur in D. Let $\mathscr{G}$ and $\mathscr{H}$ be two more new variables. We set $P'=(D', \mathscr{H})$, where D' is the union of $\bar{D}$, D_{Next}, D_{exp}, $D_{\mathscr{L}}$, D_f, plus the following two definitions:

$$\mathscr{H} = \lambda \boldsymbol{x}.\ \mathscr{G}(Next\, \boldsymbol{x}) eq^0\, x_1\, \boldsymbol{x}\, x_1,$$
$$\mathscr{G} = \lambda sez\, \boldsymbol{x}\, y.\ \textbf{if}\ \mathscr{F}(\mathscr{S}_l sez)(\mathscr{R}_l sez)(\mathscr{Z}_l sez)\boldsymbol{x}(\mathscr{L} sezy)\ \textbf{then}\ \bar{M} sez\, \boldsymbol{x}$$
$$\textbf{else}\ \mathscr{G}(\mathscr{S}_l sez)(\mathscr{R}_l sez)(\mathscr{Z}_l sez)\boldsymbol{x}\,(\mathscr{S}_l sez(\mathscr{L} sezy)).$$

We leave it to the reader to verify that P_{fin} is indeed typable in *EML* with $\mathscr{G}: \forall\alpha.\,\tilde{\alpha}\to\mathbf{0}^k\to\alpha\to\mathbf{0}$. We claim that, for all finite $(\mathscr{A}, \boldsymbol{a})$ with $|\mathscr{A}|>1$, and all $a\in\mathscr{A}$,

(*) $\quad \mathscr{A}, D\models M\boldsymbol{a}\twoheadrightarrow a$ iff $\mathscr{A}, D'\models \mathscr{H}\boldsymbol{a}\twoheadrightarrow a$.

In order to show the claim, assume for a while that (D', S, R, Z) defines arithmetic in $(\mathscr{A}, \boldsymbol{a})$ of size $m\geqslant|\mathscr{A}|$, and suppose that, for some $p<m$, we have $f(\kappa(\mathscr{A}, \boldsymbol{a}), p)\neq 0$. Then, by Lemma 5.10,

$$\mathscr{A}, D_f\models \mathscr{F}\ succ\ eq\ 0\ \boldsymbol{a}\ p\twoheadrightarrow false$$

within the bound $exp_l(m)$. Thus,

$$\mathscr{A}, D_f\models_m \mathscr{F}\ succ\ eq\ 0\ \boldsymbol{a}\ succ^p 0\twoheadrightarrow false;$$

whence,

$$\mathscr{A}, D'\models \mathscr{F}\,(\mathscr{S}_l SRZ)(\mathscr{R}_l SRZ)(\mathscr{Z}_l SRZ)\boldsymbol{a}((\mathscr{S}_l SRZ)^p(\mathscr{Z}_l SRZ))\twoheadrightarrow false.$$

Similarly, if $f(\kappa(\mathscr{A}, \boldsymbol{a}), p)=0$, then the above term reduces to *true*. In particular, if $p=\varphi(\kappa(\mathscr{A}, \boldsymbol{a}))=\mu y(f(\kappa(\mathscr{A}, \boldsymbol{a}), p)=0)$, then p is a bound for P in $(\mathscr{A}, \boldsymbol{a})$, and since $p<m$, it follows that $\mathscr{A}, D\models_m M\boldsymbol{a}\twoheadrightarrow a$, for some a; whence $\mathscr{A}, D'\models_m \bar{M}\,SRZ\boldsymbol{a}\twoheadrightarrow a$. Thus, $\mathscr{G}\,SRZ\boldsymbol{a}(S^pZ)\twoheadrightarrow a$.

Let $S_0=Next\,\boldsymbol{a}$, $R_0=eq^{\mathbf{0}}$, $Z_0=a_1$, and let for $p\geqslant 0$, $S_{p+1}=\mathscr{S}_l S_p R_p Z_p$, $R_{p+1}=\mathscr{R}_l S_p R_p Z_p$, $Z_{p+1}=\mathscr{Z}_l S_p R_p Z_p$. For each p, the tuple (D', S_p, R_p, Z_p) defines arithmetic in $(\mathscr{A}, \boldsymbol{a})$ of size $exp_l^p(|\mathscr{A}|)$. By the above consideration, and by Lemma 5.11, we obtain the following sequence of reductions with respect to $\mathscr{A}$ and D':

$$\mathscr{H}\boldsymbol{a}\twoheadrightarrow\mathscr{G}S_0R_0Z_0\twoheadrightarrow\mathscr{G}S_1R_1Z_1\boldsymbol{a}(\mathscr{L}S_0R_0Z_0Z_0)\twoheadrightarrow\mathscr{G}S_1R_1Z_1\boldsymbol{a}(S_1Z_1)$$

$$\twoheadrightarrow\mathscr{G}S_2R_2Z_2\boldsymbol{a}((S_2)^2Z_2)\twoheadrightarrow\cdots\twoheadrightarrow\mathscr{G}S_pR_pZ_p\boldsymbol{a}((S_p)^pZ_p)\twoheadrightarrow\cdots$$

provided, for all successive p, it holds that $f(\kappa(\mathscr{A}, \boldsymbol{a}), p)\neq 0$. Indeed, we have $p<exp_l^p(|\mathscr{A}|)$ and the evaluation of $\mathscr{F}\,S_{p+1}R_{p+1}Z_{p+1}\boldsymbol{a}(\mathscr{L}\,S_pR_pZ_p(S_p^pZ_p))$ to *false* can always be completed.

It follows that if $\varphi(\kappa(\mathscr{A}, \boldsymbol{a}))$ is undefined, then both sides of (*) are false. Otherwise, both $M\boldsymbol{a}$ and $\mathscr{H}\boldsymbol{a}$ have normal forms and these normal forms are equal. □

Now, we can conclude with the main result of this section.

5.13. Theorem. *For each recursive scheme with arithmetic $P=(D, M):\mathbf{0}^k\to\mathbf{0}$, there exists an equivalent EML-program. Thus, EML is a programming language of universal computational power; in particular, the spectral complexity of EML is equal to the class of all recursively enumerable sets. It follows that the halting problem of EML over finite interpretations is undecidable.*

Proof. Let $P_{fin}=(D_{fin}, M_{fin})$ be the recursive procedure of Lemma 5.12, and let $P_{inf}=(\bar{D}\cup D_{Next}, \lambda\boldsymbol{x}.\,\bar{M}(Next\,\boldsymbol{x})eq^{\mathbf{0}}x_1\boldsymbol{x})$, where $P_{Next}=(D_{Next}, Next)$ is the recursive

procedure of Lemma 3.4. Now, P_{fin} can simulate P in all finite interpretations of size at least 2, while Lemma 5.8 provides a solution for all infinite ones. Of course, one-element interpretations do not make any difficulty. We can now unify the solutions for the finite, one-element and infinite cases, obtaining one program equivalent to P in all interpretations. For this, one has to make the simulating program behave as if the interpretation were infinite until it is discovered that $Next\,a\,a$ evaluates to a, for some a. If this happens, the program runs P_{fin} (cf. the proof of Corollary 3.8). The construction details are left to the reader. □

Appendix. Syntax-oriented type-inference

Let us observe that the type-inference system for FL is *syntax-oriented* or "deterministic" in the following sense. Given a type assertion of the form $E \vdash M : \tau$, there is only one rule which could be used as the last step in deriving such an assertion (the rule being determined syntactically by the term M) and, thus, a derivation of type for a term M has a unique tree-structure. This is not the case, however, for the type-inference systems for ML and EML: these systems are "nondeterministic" because of the rules GEN and INST which may be invoked at any time, so that the strategy to follow in constructing a derivation tree is not obvious. It is sometimes more convenient to deal with syntax-oriented systems. Such a system for ML was developed in [2], and we extend this approach also for EML.

We say that a type σ is *consistent* with an environment E iff σ does not bind variables free in E. We will write $E \vdash^{1}_{ML} M : \tau$ if τ is an open type and $E \vdash_{ML} M : \tau$ can be derived with the help of the rules CONST, APP, COND, ABS, FIX and the following rules:

$$(\mathrm{LET}_1) \qquad \frac{E \vdash N : body(\sigma),\ E(x:\sigma) \vdash M : \tau}{E \vdash (\mathbf{let}\, x = N\, \mathbf{in}\, M) : \tau}, \quad \text{where } \sigma \text{ is consistent with } E;$$

$$(\mathrm{VAR}_1) \qquad E \vdash x : \tau, \qquad \text{if } (x:\sigma) \text{ is in } E \text{ and } \sigma \preceq \tau.$$

Similarly, we write $E \vdash^{1}_{EML} M : \tau$ if τ is open and $E \vdash_{EML} M : \tau$ is derived with the help of the rules CONST, APP, COND, ABS, LET_1, VAR_1 and the following rule:

$$(\mathrm{FIX}^{+}_{1}) \qquad \frac{E(x:\sigma) \vdash M : body(\sigma)}{E \vdash (\mathbf{fix}\, x.\, M) : \tau}, \quad \text{where } \sigma \preceq \tau \text{ and } \sigma \text{ consistent with } E.$$

The following is an extended form of Lemma 4.2(1) and 5.1(1).

A.1. Lemma. *Let* $\vdash$ *stand for any of the four* $\vdash_{ML}$, $\vdash_{EML}$, $\vdash^{1}_{ML}$, $\vdash^{1}_{EML}$. *Then*

(1) *If* $E(x:\sigma) \vdash M : \sigma'$ *and* $\sigma_1 \preceq \sigma$, *then* $E(x:\sigma_1) \vdash M : \sigma'$;

(2) *If* $\sigma' =_{\alpha} \sigma$ *and* $E' =_{\alpha} E$, *then* $E \vdash M : \sigma$ *implies* $E' \vdash M : \sigma'$;

(3) *If* $E \vdash M : \sigma$, *then* $E[\boldsymbol{\rho}/\boldsymbol{\alpha}] \vdash M : \sigma[\boldsymbol{\rho}/\boldsymbol{\alpha}]$.

Proof. Condition (1) is left to the reader. To show (2), note that by Lemma 4.2(1) and 5.1(1), it remains to consider $\vdash^1$, i.e. to show that if $E' =_\alpha E$ then $E \vdash^1 M : \tau$ implies $E' \vdash^1 M : \tau$, where all the types are open, and $\vdash^1$ is either $\vdash^1_{ML}$ or $\vdash^1_{EML}$. The proof goes by induction on the length of the derivation of $E \vdash^1 M : \tau$, by cases depending on the last rule used in the derivation. The base step (a single application of VAR_1) follows from the fact that for all x, if $E(x) \preceq \tau$, then also $E'(x) \preceq \tau$. In the induction steps we use the observation that $E =_\alpha E'$ implies $E(x:\sigma) =_\alpha E'(x:\sigma)$ and $FV(E) = FV(E')$.

Condition (3) is also shown by induction. We show, as an example, the induction step for FIX_1^+. Assume $E \vdash \mathbf{fix}\, x.M : \tau$ to be derived by FIX_1^+ from $E(x:\sigma) \vdash M : body(\sigma)$. By the induction hypothesis we have $E[\boldsymbol{\rho}/\boldsymbol{\alpha}](x:\sigma[\boldsymbol{\rho}/\boldsymbol{\alpha}]) \vdash M : body(\sigma)[\boldsymbol{\rho}/\boldsymbol{\alpha}]$. By (2), we can assume that σ does not bind variables free in $\boldsymbol{\rho}$ and that $\boldsymbol{\alpha}$ are not bound in σ. Thus, $body(\sigma)[\boldsymbol{\rho}/\boldsymbol{\alpha}] = body(\sigma[\boldsymbol{\rho}/\boldsymbol{\alpha}])$ and we may apply FIX_1^+ to get $E[\boldsymbol{\rho}/\boldsymbol{\alpha}] \vdash M : \tau[\boldsymbol{\rho}/\boldsymbol{\alpha}]$, because $\sigma[\boldsymbol{\rho}/\boldsymbol{\alpha}] \preceq \tau[\boldsymbol{\rho}/\boldsymbol{\alpha}]$, by Lemma 4.1(4). □

A.2. Lemma. *Let* $\vdash$ *stand for any of the two* $\vdash_{ML}$, $\vdash_{EML}$.

(1) *Assume* $E \vdash M : \sigma$. *Then* $E \vdash^1 M : body(\sigma)$.

(2) *If* $E \vdash^1 M : \tau$, *then* $E \vdash M : \tau$. *In particular, if* σ *does not bind variables free in* E *and* $E \vdash^1 M : body(\sigma)$, *then* $E \vdash M : \sigma$, *by generalization.*

Proof. (1) The proof goes by induction with respect to the derivation $E \vdash M : \sigma$. The steps for rules CONST, APP, COND, ABS, FIX are easy since all types involved here are open. An application of VAR is replaced by VAR_1 since $\sigma \preceq body(\sigma)$. For LET, assume that $E \vdash \mathbf{let}\, x = N \,\mathbf{in}\, M : \tau$ follows from $E \vdash N : \sigma$ and $E(x:\sigma) \vdash M : \tau$. By α-conversion, we may assume that σ does not bind variables free in E and, thus, $E \vdash^1 N : body(\sigma)$ and we can apply LET_1. Similarly for FIX^+. Rule GEN does not change the body of derived type; thus, this step is obvious. For INST, assume $E \vdash M : \sigma'$ derived from $E \vdash M : \sigma$, with $\sigma \preceq \sigma'$; say $\sigma = \forall \boldsymbol{\alpha}.\rho$, $\sigma' = \forall \boldsymbol{\beta}.\rho[\boldsymbol{\tau}/\boldsymbol{\alpha}]$, where $\rho = body(\sigma)$. By the induction hypothesis, $E \vdash^1 M : \rho$, and we can assume that $\boldsymbol{\alpha}$ are not free in E, by α-conversion. Thus, $E[\boldsymbol{\tau}/\boldsymbol{\alpha}] = E$ and $E \vdash M : \rho[\boldsymbol{\tau}/\boldsymbol{\alpha}]$, i.e. $E \vdash M : body(\sigma')$, by Lemma A.1(3).

(2) Clearly, one can replace every application of VAR_1 by VAR and GEN. An application of LET_1 is replaced by LET, preceded by a sequence of GENs (which turn $E \vdash N : body(\sigma)$ into $E \vdash N : \sigma$). Similarly, for FIX_1^+ we use a sequence of GENs, a FIX^+ and, finally, an INST, to get $E \vdash \mathbf{fix}\, x.M : \tau'$ from $E \vdash \mathbf{fix}\, x.M : \sigma$. □

In the sequel we often identify derivations yielding $E \vdash M : \sigma$ with $E \vdash^1 M : body(\sigma)$, assuming σ does not bind variables free in E. For uniformity, we use the notational convention that $\vdash^1_n$ stands just for $\vdash_n$.

In order to show the correctness of reductions with respect to type-inference, we first state the following fact.

A.3. Lemma. *Let* $\vdash$ *stand for any of the three* $\vdash_{ML}$, $\vdash_{EML}$, $\vdash_n$. *If* $E(x:\sigma) \vdash M : \sigma'$ *and* $E \vdash N : \sigma$, *then* $E \vdash M[N/x] : \sigma'$.

Proof. An easy induction with respect to the derivation $E(x:\sigma)\vdash M:\sigma'$. □

Now we can prove the lemmas about reductions.

A.4. Lemma (Lemmas 3.1, 4.2(3) and 5.1(3)). *Let* $\vdash$ *stand for any of the three:* $\vdash_{ML}$, $\vdash_{EML}$, $\vdash_n$. *If* $E\vdash M:\sigma$ *and* $\mathscr{A}, D\models M\twoheadrightarrow M'$, *then* $E\vdash M':\sigma$.

Proof. Clearly, it is enough to assume $M\to M'$. We proceed by induction with respect to the construction of M. Here we show the cases of application and fixpoint. We assume σ not to bind variables free in E. Let $\tau=body(\sigma)$. Then we have $E\vdash^1 M:\tau$.

Let $M\equiv PQ$. The type assertion $E\vdash^1 M:\tau$ could only be obtained from $E\vdash^1 P:\tau'\to\tau$ and $E\vdash^1 Q:\tau'$. If $M'\equiv P'Q'$ and $P\to P'$ or $Q\to Q'$, then by the induction hypothesis $E\vdash^1 P':\tau'\to\tau$ and $E\vdash^1 Q':\tau'$; whence, $E\vdash^1 M':\tau$. Since $\tau=body(\sigma)$, we also have $E\vdash M':\sigma$. The case of M being a δ-redex is even easier. Thus, assume that $M\equiv(\lambda x.R)Q$ and $M'\equiv R[Q/x]$. The typing $E\vdash^1(\lambda x.R)Q:\tau'\to\tau$ must follow from $E(x:\tau')\vdash^1 R:\tau$. Use Lemma A.2 to get $E\vdash R[Q/x]:\tau$ which also means that $E\vdash^1 R[Q/x]:\tau$.

Now assume that $M\equiv\mathbf{fix}\,x.P$. The only nonobvious case is when $M'\equiv P[\mathbf{fix}\,x.P/x]$. For FL and ML we use Lemma A.2, as above. For EML, $E\vdash^1 M:\tau$ must follow from $E(x:\sigma')\vdash^1 M:body(\sigma')$, for some $\sigma'\preceq\tau$. It follows that $E(x:\sigma')\vdash M:\sigma'$; whence, $E\vdash\mathbf{fix}\,x.P:\sigma'$ and then $E\vdash P[\mathbf{fix}\,x.P/x]:\sigma'$. By INST, we get $E\vdash P[\mathbf{fix}\,x.P/x]:\tau$, whence $E\vdash P[\mathbf{fix}\,x.P/x]:\sigma$. □

In a syntax-oriented type-inference system we can assume that in a derivation of the form "$E\vdash M:\tau$", each subterm of M is assigned a type exactly once (provided we count different occurrences of the same term as different subterms). Thus, we can think of a derivation as a "typing", i.e. a function which assigns (open) types to all subterms of M.

A.5. Lemma (Lemmas 3.2, 4.3 and 5.2). *Let* $L\in\{FL_n, ML, EML\}$, *where* $n\in\omega$ *and let* (D, M) *be a semiprogram. Assume* (D', M') *to be constructed from* (D, M) *by Transformation I, so that the strategy in Case* 2 *is as follows: Method* 1 *for ML, Method* 2 *otherwise. The following are equivalent:*

(a) $E\vdash_{\mathrm{L}}(D, M):\mathbf{0}^k\to\mathbf{0}$, *for some* E;

(b) $E'\vdash_{\mathrm{L}}(D', M'):\mathbf{0}^k\to\mathbf{0}$, *for some* E'.

Proof. For each of Steps 1, 2, we have to consider the three cases for the different typing systems. We assume the notation from Section 2, and we also use α-conversion, both on object terms and on types. In particular, our terms are always assumed α-correct.

We stick to the notation used in the definition of Transformation I of Section 2. The proof is by induction on the number of steps in the transformation, i.e. we show the hypothesis for an arbitrary single step.

Step 1(a). **(fix)**, (a)⇒(b): Consider the derivations which yield the typing for (D, M), and consider the type assertions for $C \equiv \mathbf{fix}\, x.\, P$.

Case FL_n: Here C is assigned type by an application of rule FIX, i.e. we have

$$(*) \qquad E \cup A \vdash \mathbf{fix}\, x.\, P : \tau$$

derived from $E(x:\tau) \cup A \vdash P : \tau$, where A assigns types to the variables $\boldsymbol{y}$. Assume $A(y_i) = \rho_i$. Let $E' = E \cup \{X : \boldsymbol{\rho} \to \tau\}$. One can easily see that

$$(**) \qquad E' \cup A \vdash X\boldsymbol{y} : \tau;$$

whence, $E' \cup A \vdash P[X\boldsymbol{y}/x] : \tau$, which gives $E' \vdash \lambda \boldsymbol{y}.\, P[X\boldsymbol{y}/x] : \boldsymbol{\rho} \to \tau$. Thus, the new definition is well-typed. Also, the derivation using $(*)$ may be modified by replacing $(*)$ by $(**)$.

One remark is necessary here: the free variables of C are either defined in D or are bound outside by λ's (since C is outermost). If the initial semiprogram is of level n, then the types $\boldsymbol{\rho}$ assigned to $\boldsymbol{y}$ must be of level at most $n-1$. Thus, the new type $\boldsymbol{\rho} \to \tau$ is still of level n, and the transformed semiprogram is still in FL_n.

Case ML: We may assume that a type for C is obtained by an application of the rule FIX_1, i.e. $E \cup A \vdash^1 \mathbf{fix}\, x.\, P : \tau$ is derived from $E(x:\tau) \cup A \vdash^1 P : \tau$. As in the previous case we observe that all variables $\boldsymbol{y}$ are λ-bound outside of C. Thus, all types $\rho_i = A(y_i)$ are open. We take $E' = E \cup \{X : \boldsymbol{\rho} \to \tau\}$, and we proceed as in the previous case.

Case EML: Now, C is typed with the help of the rule FIX_1^+: an assertion $E \cup A \vdash^1 \mathbf{fix}\, x.\, P : \tau'$ is derived from $E(x:\sigma) \cup A \vdash^1 P : body(\sigma)$. Let $\tau = body(\sigma)$. Again, $A(y_i) = \rho_i$ are open types. Let $E' = E \cup \{X : \forall.\, \boldsymbol{\rho} \to \tau\}$. Using GEN and INST, we can derive $E' \cup A \vdash X\boldsymbol{y} : \sigma$ and also $E' \cup A \vdash X\boldsymbol{y} : \tau'$; whence, also $E' \cup A \vdash^1 X\boldsymbol{y} : \tau'$. On the other hand, $E' \cup A \vdash P[X\boldsymbol{y}/x] : \sigma$, by Lemma A.3; whence, $E' \vdash \lambda \boldsymbol{y}.\, P[X\boldsymbol{y}/x] : \boldsymbol{\rho} \to \tau$ and, finally, $E' \vdash \lambda \boldsymbol{y}.\, P[X\boldsymbol{y}/x] : \forall.\, \boldsymbol{\rho} \to \tau\}$.

Step 1(b). **(fix)**, (b)⇒(a): Let E be E' without the typing for X. Note that X occurs only once in (D', M'), except possibly in its own definition. The context of its occurrence is $X\boldsymbol{y}$.

Case FL_n: Consider first the new definition "$X = \lambda \boldsymbol{y}.\, P[X\boldsymbol{y}/x]$". It must be the case that $E'(X) = \boldsymbol{\rho} \to \tau$ and $E' \vdash \lambda \boldsymbol{y}.\, P[X\boldsymbol{y}/x] : \boldsymbol{\rho} \to \tau$. For $A(y_i) = \rho_i$, we have $E \cup A \vdash X\boldsymbol{y} : \tau$ and $E \cup A \vdash P[X\boldsymbol{y}/x] : \tau$. Since X occurs in the latter derivation only in the context $X\boldsymbol{y}$, always assigned the type τ, we also have $E(x:\tau) \cup A \vdash P : \tau$; whence, $E \cup A \vdash \mathbf{fix}\, x.\, P : \tau$.

Now take the only occurrence of $X\boldsymbol{y}$ outside the definition of X. An appropriate derivation uses a type assertion $E' \cup B \vdash X\boldsymbol{y} : \tau'$. However, since $X : \boldsymbol{\rho} \to \tau$, we see that $\tau' = \tau$ and $B(y_i) = \rho_i$, i.e. $B = A$ (recall that C was outermost and, thus, $\boldsymbol{y}$ must be λ-bound). Thus, we can replace the above by $E \cup A \vdash \mathbf{fix}\, x.\, P : \tau$ in the derivation.

Case ML: The proof is similar to the above (use $\vdash^1$ instead of $\vdash$).

Case EML: Consider the definition "$X = \lambda \boldsymbol{y}.\, P[X\boldsymbol{y}/x]$". Now we have $E' \vdash \lambda \boldsymbol{y}.\, P[X\boldsymbol{y}/x] : \forall.\, \boldsymbol{\rho} \to \tau$ and $E'(X) = \forall.\, \boldsymbol{\rho} \to \tau$ (note that the type of X must be closed).

Let $A(y_i)=\rho_i$ (again $\boldsymbol{\rho}$ are open). We have $E'\vdash^1 \lambda \boldsymbol{y}.P[X\boldsymbol{y}/x]:\boldsymbol{\rho}\to\tau$ and $E'\cup A\vdash^1 P[X\boldsymbol{y}/x]:\tau$. On the other hand, $E'\cup A\vdash^1 X\boldsymbol{y}:\tau$.

Let α be all type variables of τ not free in $\boldsymbol{\rho}$. Then $E'\cup A\vdash P[X\boldsymbol{y}/x]:\forall\alpha.\tau$ and $E'\cup A\vdash X\boldsymbol{y}:\forall\alpha.\tau$. All types derived for $X\boldsymbol{y}$ in $E'\cup A$ must be instances of $\forall\alpha.\tau$, because types of $\boldsymbol{y}$ are open, i.e. no type variable free in $\boldsymbol{\rho}$ may be instantiated for the application. Thus, as in the previous cases, $E'(x:\forall\alpha.\tau)\cup A\vdash P:\forall\alpha.\tau$ and we conclude that

(*) $\quad E'\cup A\vdash \mathbf{fix}\, x.P:\forall\alpha.\tau.$

Now turn to the only occurrence of $X\boldsymbol{y}$ outside the definition of X. In the appropriate derivation (in the syntax-oriented system) we have $E'\cup B\vdash^1 X\boldsymbol{y}:\mu$, for some open μ. Also, $B(y_i)=\pi_i$ may be assumed to be open. Since $E'(X)=\forall.\boldsymbol{\rho}\to\tau$, we have $\forall.\boldsymbol{\rho}\to\tau\preceq\boldsymbol{\pi}\to\tau$. Applying the corresponding substitution to the type assertion (*) we get $E\cup B\vdash \mathbf{fix}\, x.P:\forall\boldsymbol{\alpha}.\tau'$, for an appropriate τ', and by instantiation $E\cup B\vdash \mathbf{fix}\, x.P:\mu$. Since μ is open, $E\cup B\vdash^1 \mathbf{fix}\, x.P:\mu$, and our occurrence of $X\boldsymbol{y}$ may be replaced by $\mathbf{fix}\, x.P$.

Step 1(b). (let):

For Step 1(b) we show only the case for *ML*. The two other cases are handled similarly as in Step 1(a).

(a)⇒(b): An appropriate derivation (in the syntax-oriented system) will contain the following type assertion: $E\cup A\vdash^1(\mathbf{let}\, x=P\,\mathbf{in}\, Q):\tau$. It must follow by LET_1 from $E\cup A\vdash^1 P:body(\sigma)$ and $E(x:\sigma)\cup A\vdash^1 Q:\tau$. Types $\rho_i=A(y_i)$ must be open. In the derivation for $E(x:\sigma)\cup A\vdash^1 Q:\tau$ the rule VAR_1 is used for x once for every occurrence of x in Q. Assume that the type derived for $x^{(j)}$ is $\tau_j\succeq\sigma$. Let $E'=E\cup\{X_j:\boldsymbol{\rho}\to\tau_j\mid j=1,\ldots,n\}$. One can easily see that $E\cup A\vdash X_j\boldsymbol{y}:\tau_j$; thus, replacing $x^{(j)}$ by $X_j\boldsymbol{y}$ in Q yields a term of type τ in the environment $E'\cup A$. Also, the new definitions are well-typed since we have $E'\vdash^1\lambda\boldsymbol{y}.P:\boldsymbol{\rho}\to body(\sigma)$ and, since σ may be assumed not to bind variables free in E, we get $E'\vdash\lambda\boldsymbol{y}.P:\boldsymbol{\rho}\to\tau_j$, by INST.

(b)⇒(a): E is E' without the type assumptions for X_j. The term C' which replaces C has only one occurrence of $X_j\boldsymbol{y}$, for each j. The type derived (in the syntax-oriented system) for X_j is of the form $\boldsymbol{\rho}\to\tau_j$ (the argument types $\boldsymbol{\rho}$ are always the same because of the λ-bindings of $\boldsymbol{y}$). Thus, we have $E'(X_j)=\boldsymbol{\rho}\to\tau$. From the definitions of X_j we also have $E\cup A\vdash P:\tau_j$, where $A(y_i)=\rho_i$. Take the principal type σ of P in $E\cup A$. Clearly, if $E\cup A\vdash C':\tau'$, then also $E(x:\sigma)\cup A\vdash Q:\tau'$, since differently instantiated occurrences of x may replace $X_j\boldsymbol{y}$. Since $E\cup A\vdash P:\sigma$, we have $E\cup A\vdash\mathbf{let}\, x=P\,\mathbf{in}\, Q:\tau'$.

Step 2(λ).

We sketch the proof for all the cases together.

(a)⇒(b): We have $E\cup A\vdash^1\lambda x.P:\tau\to\rho$ obtained from $E(x:\tau)\cup A\vdash^1 P:\rho$. Let $A(y_i)=\rho_i$. These types must be open (of level $n-1$) as $\boldsymbol{y}$ are λ-bound. Thus $E\vdash^1\lambda\boldsymbol{y}x.P:\boldsymbol{\rho}\to\tau\to\rho$ and this enables us to set $E'(F)=\boldsymbol{\rho}\to\tau\to\rho$ (for *FL* and *ML*) or

$E'(F)=\forall.\rho\to\tau\to\rho$ (for *EML*). After substituting Fy for $\lambda x.P$ the typing remains correct, since $E\cup A\vdash Fy:\tau\to\rho$.

(b)⇒(a): The new definition is $F=\lambda yx.P$ and, thus, it must be the case that $E'(F)=\rho\to\tau\to\rho$ (for *FL* and *ML*) or $E'(F)=\forall.\rho\to\tau\to\rho$ (for *EML*). Thus, for $A(y_i)=\rho_i$ we have $E(x:\tau)\cup A\vdash^1 P:\rho$ and $E\cup A\vdash^1\lambda x.P:\tau\to\rho$. Take the only occurrence of Fy outside of P. This is typed by $E'\cup B\vdash^1 Fy:\mu$. Types in B may be assumed open, so in *FL* and *ML* we have $B=A$ and $E'\cup A\vdash^1 Fy:\tau\to\rho$.

In *EML* we have B obtained from A by substitution (as for **fix**). If $B(y_i)=\pi_i$, then $E'\vdash F:\boldsymbol{\pi}\to\mu$ and $\boldsymbol{\pi}\to\mu\geq\forall.\rho\to\tau\to\rho$. Since $E\cup A\vdash^1\lambda x.P:\tau\to\rho$, by applying the substitution we get $E\cup B\vdash^1\lambda x.P:\mu$, and we can replace Fy by $\lambda x.P$ with no loss of typing. □

A.6. Lemma (Lemmas 3.2 and 5.2): *Let* $L\in\{FL_n, EML\}$, *and let* (D, M) *be a semi-program. Assume* $(\emptyset, M_D)$ *to be constructed from* (D, M) *by Transformation II. The following are equivalent:*

(a) $E\vdash_L(D, M):\mathbf{0}^k\to\mathbf{0}$, *for some* E;

(b) $E'\vdash_L M_D:\mathbf{0}^k\to\mathbf{0}$, *for some* E'.

Proof. It suffices to show that if (D', M') is obtained from (D, M) by one step, then (a) is equivalent to

(b') $E\vdash_L(D', M'):\mathbf{0}^k\to\mathbf{0}$, for some E'.

(a)⇒(b'): We have $E\vdash T:\sigma$, for some $\sigma=E(F)$. Let $E'=E-\{F:\sigma\}$. Then $E'(F:\sigma)\vdash T:\sigma$; whence, $E'\vdash\mathbf{fix}\,F.T:\sigma$. Thus, for each N, if $E\vdash N:\rho$, then $E'\vdash\mathbf{let}\,F=\mathbf{fix}\,F.T\,\mathbf{in}\,N:\rho$ and the typing remains correct after the modification.

(b')⇒(a): We have $E'\vdash\mathbf{let}\,F=\mathbf{fix}\,F.T\,\mathbf{in}\,M:\mathbf{0}^k\to\mathbf{0}$, and similarly for the right-hand sides of the definitions. This means that, for some σ, $E'(F:\sigma)\vdash M:\mathbf{0}^k\to\mathbf{0}$ and $E'(F:\sigma)\vdash T:\sigma$. It suffices to put $E'=E\cup\{F:\forall.\sigma\}$. □

References

[1] S. Brown, D. Gries and T. Szymanski, Program schemes with push-down stores, *SIAM J. Comput.* **1** (3) (1972) 242–268.

[2] D. Clément, J. Despeyroux, T. Despeyroux and G. Kahn, A simple applicative language: Mini-ML, in: *Proc. ACM Conference on Lisp and Functional Programming*, (1986) 13–27.

[3] R.L. Constable and D. Gries, On classes of program schemata, *SIAM J. Comput.* **1** (1) (1972) 66–118.

[4] L. Damas and R. Milner, Principal type schemes for functional programs, in: *Proc. 9th ACM Symp. Principles of Programming Languages* (1982) 207–212.

[5] W. Damm, The IO- and OI-hierarchies, *Theoret. Comput. Sci.* **20** (1982) 95–206.

[6] W. Damm, E. Fehr and K. Indermark, Higher type recursion and self-application as control structures, in: D. Neuhold, ed., *Formal Description of Programming Concepts* (North-Holland, Amsterdam, 1978) 461–487.

[7] W. Damm and A. Goerdt, An automata-theoretic characterization of the OI-hierarchy, *Inform. and Control* **71** (1986) 1–32.

[8] J. Engelfriet, Iterated push-down automata and complexity classes, in: *Proc. 15th STOC* (1983) 365–373.

[9] J. Engelfriet and E.M. Schmidt, IO and OI, *J. Comput. System Sci.* **15** (1977) 328–353 and **16** (1978) 67–99.

[10] H. Friedman, Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory, in: *Logic Colloquium* '69 (North Holland, Amsterdam, 1971) 361–389.

[11] A. Goerdt, On the computational power of the finitely typed lambda-terms, in: *Proc. MFCS* (1988) 318–328.

[12] A. Goerdt, Characterizing complexity classes by general recursive definitions in higher types, 1989, to appear.

[13] F. Henglein, Type inference and semi-unification, in: *Proc. ACM Symp. LISP and Functional Programming* (1988) 184–197.

[14] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation* (Addison-Wesley, Reading, MA, 1979).

[15] K. Indermark, Schemes with recursion on higher types, in: *Proc. of 5th MFCS*, Lecture Notes in Computer Science, Vol. 45 (Springer, Berlin, 1976) 352–358.

[16] A.J. Kfoury, A.P. Stolboushkin and P. Urzyczyn, Some open problems in the theory of program schemes and dynamic logics, (Russian), *Uspekhi Mat. Nauk*, **44** (1989) 35–55.

[17] A.J. Kfoury, J. Tiuryn and P. Urzyczyn, The hierarchy of finitely typed functional programs, in: *Proc. 2nd IEEE Symp. Logic in Computer Science* (1987) 225–235.

[18] A.J. Kfoury, J. Tiuryn and P. Urzyczyn, On the computational power of universally polymorphic recursion, in: *Proc. 3rd IEEE Symp. Logic in Computer Science* (1988) 72–81.

[19] A.J. Kfoury, J. Tiuryn and P. Urzyczyn, Computational consequences and partial solutions of a generalized unification problem, in: *Proc 4th IEEE Symp. Logic in Computer Science* (1989) 98–105.

[20] A.J. Kfoury, J. Tiuryn and P. Urzyczyn, Type-checking in the presence of polymorphic recursion, Research Report, Boston University, 1989.

[21] A.J. Kfoury and P. Urzyczyn, Necessary and sufficient conditions for the universality of programming formalisms, *Acta Inform.* **22** (1985) 347–377.

[22] A.J. Kfoury and P. Urzyczyn, Finitely typed functional programs, Part I: syntax, semantics and implementation, Research Report, Boston University, 1986; Part II: Comparisons to imperative languages, Research Report, Boston University, 1988.

[23] A.J. Kfoury and P. Urzyczyn, ALGOL-like languages with higher-order procedures and their expressive power, in: *Proc. Logic at Botik* '89, Lecture Notes in Computer Science, Vol. 363 (Springer, Berlin, 1989) 186–199.

[24] J. Klop, *Combinatory Reduction Systems*, Mathematical Centre Tracts **127** (Mathematical Centre, Amsterdam, 1980).

[25] W. Kowalczyk, D. Niwiński and J. Tiuryn, A generalization of Cook's auxiliary-pushdown-automata theorem, Research Report, Institute of Math, Univ of Warsaw, 1986.

[26] D. Kozen and J. Tiuryn, Logics of programs, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B* (Elsevier, Amsterdam, 1990) 789–840.

[27] H. Langmaack, On procedures as open subroutines, *Acta Inform.* **2** (1973) 311–333 and **3** (1974) 227–241.

[28] H. Leiss, On type inference for object-oriented programming languages, in: *Proc. Logik in der Informatik* (Springer-Verlag, Karlsruhe, 1987) 151–172.

[29] R. Milner, A theory of type polymorphism in programming, *J. Comput. System Sci.* **17** (1978) 348 375.

[30] A. Mycroft, Polymorphic type schemes and recursive definitions, in: M. Paul and B. Robinet, eds., *Internat. Symp. on Programming*, Lecture Notes in Computer Science, Vol. 167 (Springer, Berlin, 1984) 217–228.

[31] M.S. Paterson and C.E. Hewitt, Comparative schematology, in: *Record on Project MAC Conference on Concurrent Systems and Parallel Computation* (ACM, New York, 1970) 119–128.

[32] S.L. Peyton Jones, *The Implementation of Functional Programming Languages* (Prentice-Hall, Englewood Cliffs, NJ, 1987).

[33] G. Plotkin, LCF considered as a programming language, *Theoret. Comput. Sci.* **5** (1977) 223–255.

[34] G. Plotkin, A structural approach to operational semantics, Report DAIMI FN-19, Computer Science Dept, Aarhus Univ, 1981.

[35] J. Tiuryn, Higher-order arrays and stacks in programming: An application of complexity theory to logics of programs, in: *Proc. 12th MFCS*, Lecture Notes in Computer Science, Vol. 233 (Springer, Berlin, 1986) 177–198.

[36] J. Tiuryn and P. Urzyczyn, Some connections between logics of programs and complexity theory, *Theoret. Comput. Sci.* **60** (1988) 83–108.

[37] P. Urzyczyn, A necessary and sufficient condition in order that a Herbrand interpretation be expressive relative to recursive programs, *Inform. and Control* **56** (3) (1983) 212–219.

[38] P. Urzyczyn, A remark on the expressive power of polymorphism, in: *Abstracts V-th Congress LMPS*, **5** (Nauka, Moscow, 1987) 184–188.